Henrique Manuel Mota de Azevedo

Licenciado em Engenharia Informática

Otimização da implementação em GPUs de um algoritmo de simulação de materiais granulares

Dissertação para obtenção do Grau Mestre em
Engenharia Informática

Orientador: Prof. Doutor Paulo Afonso Lopes,
Prof. Auxiliar, DI/FCT/UNL

Coorientador: Prof. Doutor Mário Vicente da Silva,
Prof. Auxiliar, DEC/FCT/UNL

Júri:

Presidente: Doutora Margarida Paula Neves Mamede

Arguentes: Doutor André Teixeira Bento Damas Mora

Vogais: Doutor Paulo Orlando Reis Afonso Lopes

FACULDADE DE CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

Dezembro, 2014

Otimização da implementação em GPUs de um algoritmo de simulação de materiais granulares.

Agradecimentos

À Faculdade de Ciências e Tecnologia que me formou e enriqueceu o meu intelecto numa área tão vasta e complexa, Engenharia Informática, através dos seus excelentes recursos humanos e logísticos necessários para a prossecução da sua missão.

Aos meus orientadores, Professor Doutor Paulo Afonso Lopes e Professor Doutor Mário Vicente da Silva que, de forma muito prestável e disponível, me aconselharam, esclareceram e conduziram na realização deste trabalho.

À minha esposa e à minha filha, Vera e Matilde, que apesar da minha ausência estiveram sempre presentes e deram-me o seu apoio na concretização e finalização deste projeto.

A todos os que não referindo nominalmente, mas que de forma direta ou indireta, contribuíram para a realização deste trabalho.

Obrigado.

Resumo

As simulações que pretendam modelar fenómenos reais com grande precisão em tempo útil exigem enormes quantidades de recursos computacionais, sejam estes de processamento, de memória, ou comunicação. Se até há pouco tempo estas capacidades estavam confinadas a grandes supercomputadores, com o advento dos processadores *multicore* e GPUs *manycore* os recursos necessários para este tipo de problemas estão agora acessíveis a preços razoáveis não só a investigadores como aos utilizadores em geral.

O presente trabalho está focado na otimização de uma aplicação que simula o comportamento dinâmico de materiais granulares secos, um problema do âmbito da Engenharia Civil, mais especificamente na área da Geotecnia, na qual estas simulações permitem por exemplo investigar a deslocação de grandes massas sólidas provocadas pelo colapso de taludes. Assim, tem havido interesse em abordar esta temática e produzir simulações representativas de situações reais, nomeadamente por parte do CGSE (*Australian Research Council Centre of Excellence for Geotechnical Science and Engineering*) da Universidade de *Newcastle* em colaboração com um membro da UNIC (Centro de Investigação em Estruturas de Construção da FCT/UNL) que tem vindo a desenvolver a sua própria linha de investigação, que se materializou na implementação, em CUDA, de um algoritmo para GPUs que possibilita simulações de sistemas com um elevado número de partículas.

O trabalho apresentado consiste na otimização, assente na premissa da não alteração (ou alteração mínima) do código original, da supracitada implementação, de forma a obter melhorias significativas tanto no tempo global de execução da aplicação, como no aumento do número de partículas a simular. Ao mesmo tempo, valida-se a formulação proposta ao conseguir simulações que refletem, com grande precisão, os fenómenos físicos. Com as otimizações realizadas, conseguiu-se obter uma redução de cerca de 30% do tempo inicial cumprindo com os requisitos de correção e precisão necessários.

Palavras-chave: Computação de Elevado Desempenho, Simulação de Partículas, GPU, CUDA.

Abstract

Accurate simulations of real-time phenomena require large amounts of computational resources (processors, memory and network infrastructures) something that, until recently, was only available through the use of supercomputers. Today, with *multicore* CPUs and *manycore* GPUs available at reasonable prices, the resources needed for this type of problems are now accessible to both researchers and general users.

This work focuses on the optimization of an application that simulates the dynamic behavior of granular materials, a Civil Engineering problem; these simulations may be used, for example, to investigate the movement of large solid masses that results from slopes collapsing. Therefore, producing simulations that mimic the behavior of real world events is a problem that is being actively researched by the CGSE (Australian Research Council Centre of Excellence for Geotechnical Science and Engineering) from the University of Newcastle in collaboration with a member of UNIC (Center Research in Structures Construction FCT / UNL), who has developed a CUDA implementation of an algorithm for GPUs that allows simulations with a large number of particles.

In this work, we performed a string of optimizations (with no or minimal modifications to the original code) on the existing implementation in order to get both a significant reduction in the application's runtime, and an increase in the problem size (number of particles in the simulation). At the same time, we were able to validate the proposed formulation through simulations that closely mimic the physical world phenomena. The optimized implementation has achieved a 30% reduction of the initial time while complying with the requirements of correctness and precision needed.

Keywords: HPC, Particle Simulation, GPU, CUDA

Glossário e lista de siglas

ALU: Arithmetic Logic Unit
CPU: Central Processing Unit
CISC: Complex Instruction Set Computing
I/O: Input/output
ECC: Error Correcting Code
EPIC: Explicitly parallel instruction computing
FPU: Floating-Point Unit
GPU: Graphics Processing Unit
GPC: Graphics Processing Clusters
NUMA: Non-Uniform Memory Access
MIMD: Multiple Instruction, Multiple Data
MISD: Multiple Instruction, Single Data
PTX: Parallel Thread eXecution ()
RISC: Reduced Instruction Set Computing
SIMD: Single Instruction, Multiple Data
SIMT: Single Instruction Multiple Thread
SISD: Single Instruction, Single Data
SP: Streaming Processor
SM: Streaming Multiprocessor
SMP: Symmetric Multiprocessor
TPC: Texture/Processing Clusters
VLIW: Very Long Instruction Word

Índice

Índice de Figuras

Índice de Tabelas

1

1 Introdução

As simulações que pretendam modelar fenómenos mecânicos complexos em tempo útil e com grande precisão, exigem invariavelmente enormes quantidades de recursos computacionais, sejam estes de processamento, de memória, ou comunicação. Se até há pouco tempo estas capacidades estavam confinadas a grandes supercomputadores, com o advento dos processadores *multicore* e GPUs *manycore* programáveis (os designados GP-GPUs que doravante abreviamos simplesmente por GPUs), disponíveis atualmente a preços razoáveis, os recursos necessários para este tipo de problemas estão acessíveis aos investigadores e utilizadores em geral. Contudo, obter o máximo rendimento dos dispositivos não depende só dos avanços tecnológicos que conduzem à miniaturização, mas também do redesenho da arquitetura dos componentes e dos modelos de execução dos programas [1]. Assiste-se então, na sequência dos avanços tecnológicos que conduziram às tecnologias *multicore* e *manycore*, a uma mudança na programação, do paradigma sequencial para o paralelo, pois só este último permite utilizar eficientemente uma fração apreciável dos recursos disponibilizados por essas arquiteturas.

1.1 Motivação

O trabalho aqui abordado é multidisciplinar, envolvendo conhecimentos nas áreas Engenharia Civil e Informática, e consiste na análise e otimização de uma aplicação de simulação do comportamento dinâmico de materiais granulares secos. A aplicação em causa, uma implementação paralela *naïve* [2] que usa a linguagem C/CUDA e GPUs, baseia-se numa num método implícito [3] [4] [5] que, sendo computacionalmente mais exigente do que os métodos explícitos geralmente utilizados, tem a vantagem de ser incondicionalmente estável.

Para evidenciar a necessidade de otimização da aplicação atual basta referir que numa estação de trabalho equipada com um GPU NVIDIA C2075/6GB (configuração máxima), uma simulação de 1M partículas demora aproximadamente 5 dias.

1.2 Abordagens computacionais para simulações com partículas

Até aos finais do século passado realizavam-se simulações com partículas usando computadores dotados de processadores convencionais, com uma lógica de programação sequencial e que, atendendo às limitações dessas arquiteturas, só eram úteis quando os dados eram relativamente pequenos. Mas para conjunto de dados subdimensionados, relativamente à realidade que se pretendia simular, não se conseguia aferir a correção dos modelos ou os resultados obtidos eram pouco significativos face às necessidades houve, para este tipo de problemas, necessidade de lhes atribuir significativamente mais recursos computacionais.

Uma das primeiras abordagens foi decompor os problemas em subcomponentes e atribuí-los, para efeito de cálculo, a processadores independentes, com a utilização das respetivas memórias para guardar os dados e um modelo de programação baseado em mensagens para comunicação e sincronização [6] [7]. As diferentes abordagens revelaram uma série de aspetos fulcrais para o desempenho e facilidade de programação: latências, arquiteturas (memória distribuída vs. partilhada), modelos de programação, bibliotecas, etc. [8] [9]. Na produção dos modelos de simulação foram tidos em conta a correção, a precisão e validação dos resultados produzidos, que se traduziu em acréscimos de recursos computacionais e a utilização de estruturas de dados complexas e eficientes [10].

Surgem então arquiteturas *Multiple Instruction, Multiple Data* (MIMD), representadas tanto por supercomputadores, como o *Cray XT3*, como por *clusters* de servidores comuns, ambas capazes de suportar aplicações paralelas baseadas em comunicações por mensagens, como as que usam bibliotecas MPI [11] [12]. Por exemplo, em meados de 2007 num servidor comum faziam-se simulações abaixo das 500.000 partículas, enquanto que com um Cray XT3 de 192 processadores, esse número atingia os 122 milhões [13].

Muito recentemente têm surgido algumas linhas de investigação no campo da simulação da interação de objetos com materiais granulares [14] [15] e simulações em tempo real com materiais granulares [16] com recurso à utilização de GPUs, que demonstraram bons desempenhos quando comparados com as arquiteturas baseadas em CPUs.

1.3 A realização atual

O presente trabalho está centrado na otimização de uma aplicação paralelizada para GPUs que simula o comportamento dinâmico de materiais granulares secos. Entenda-se por materiais granulares secos, partículas de dimensões semelhantes e formas aproximadamente esféricas, que estão em contacto entre si; no âmbito da Engenharia Civil, mais especificamente na área da Geotecnia, estas simulações permitem, por exemplo, investigar a deslocação de grandes massas sólidas provocadas pelo colapso de taludes, ou verificar a estabilidade de trabalhos de escavações, ancoragens, cravação de estacas, etc.

Assim, tem havido interesse em abordar esta temática e produzir simulações

representativas de situações reais, tendo vários estudos, nomeadamente do CGSE (*Australian Research Council Centre of Excellence for Geotechnical Science and Engineering*) da Universidade de *Newcastle* em colaboração com um membro da UNIC (Centro de Investigação em Estruturas de Construção da FCT/UNL), que tem vindo a desenvolver a sua própria linha de investigação [2] [3] [4], sendo que esta se materializou na implementação de um algoritmo para GPUs que possibilita simulações em tempo útil de sistemas com um elevado número de partículas [5].

O problema envolve assim o cálculo das sucessivas posições das partículas no plano tridimensional, com a inclusão da componente tempo, o que permite verificar a variação do seu comportamento dinâmico. Para efeitos de cálculo consideram-se as partículas como esferas de raios semelhantes, sendo que as irregularidades na sua geometria são modeladas com a introdução de forma artificial de uma resistência ao rolamento. Os potenciais contactos das partículas são determinados no início de cada passo da simulação por um algoritmo que é executado na CPU, tendo em conta o centro das partículas e o seu movimento de rotação e assumindo-se que não existem interpenetrações nem deformações das partículas. Para cada passo têm-se em consideração a posição inicial da partícula, a sua velocidade e aceleração, a massa e a força.

A formulação do problema baseia-se no deslocamento das partículas, limitado que está pelos contactos potenciais a que estão sujeitas; para tal formula-se o problema recorrendo a técnicas de programação matemática não linear. Para a resolução do problema é utilizado um algoritmo iterativo denominado por ADMM (*Alternating Direction Method of Multipliers*) [17], que opera sobre um grande volume de dados, uma vez que tipicamente o número de partículas é da ordem dos milhões, determinam-se mínimos locais e globais, atualizam-se as posições e realizam-se testes de convergência [18] [19].

A referida implementação foi desenvolvida em CUDA sobre GPUs da NVIDIA, sendo que a parte sequencial do algoritmo tem uma expressão muito reduzida. A implementação foi testada com os recursos computacionais existentes nos departamentos de Engenharia Civil e de Informática sendo que, para a simulação de 1M de partículas são necessários quase 5 dias.

1.4 Objetivos e Contribuições

Na presente dissertação propusemo-nos otimizar o supracitado algoritmo de simulação de materiais granulares, de forma a reduzir substancialmente o tempo de execução e, se possível no tempo de que dispomos, aumentar o volume de dados (número de partículas a simular), estudar com detalhe a arquitetura GPU da NVIDIA bem como a linguagem proprietária CUDA desenvolvida pelo fabricante.

Compreendendo que o problema é complexo e carece de conhecimentos profundos das arquiteturas envolvidas, dos seus modelos de execução, e da eventual necessidade de uma

“programação de baixo nível”/alta eficiência, admite-se à partida a existência de espaço para introduzir várias otimizações e obter melhorias significativas nos resultados produzidos. Colocam-se alguns desafios, como introduzir concorrência entre os *kernels*, melhorar a sua programação, minimizar a transferência de dados entre as várias memórias existentes na hierarquia, reduzir a latência, realizar uma gestão eficiente da memória, determinar a possibilidade de utilizar operações com precisão simples.

A principal contribuição desta dissertação decorre diretamente do objetivo, a otimização de um algoritmo para GPU: conseguiu-se uma redução de tempo de execução, em cerca de 30%, das simulações realizadas. Como contribuições adicionais, referimos a validação da formulação proposta ao conseguir simulações que refletem, com grande precisão, os fenómenos físicos; a validação da aplicabilidade das ferramentas utilizadas no processo de otimização por refinamentos sucessivos, e fazemos uma apreciação qualitativa do esforço necessário para otimizar este tipo de algoritmos; comprovamos ainda que é possível usar uma infraestrutura computacional baseada numa arquitetura "*deskstation*" comum equipada com um acelerador (GPU) para resolver em tempo útil problemas desta natureza, mas que pode ser necessário despender um maior esforço na paralelização/redesenho dos algoritmos e estruturas de dados.

1.5 Organização da dissertação

No Capítulo 1 faz-se uma breve introdução ao tema desta dissertação, refere-se a motivação na Secção 1.1, são descritas algumas abordagens computacionais desenvolvidas até aos nossos dias na Secção 1.2, descreve-se o problema a resolver na Secção 1.3 e os objetivos a atingir na Secção 1.4.

No Capítulo 2, abordam-se três temas: a Computação de Elevado Desempenho (HPC), a Computação com GPUs e as Otimizações de computações em GPUs. Na Secção 2.1 descrevem-se as arquiteturas existentes, os modelos de programação, curvas de aprendizagem e produtividade, e expectativas de desempenho em Computação de Alto Desempenho. Na secção 2.2, são abordadas temáticas relativas às GPUs, nomeadamente: arquiteturas existentes, modelos de programação, curvas de aprendizagem, e expectativas de desempenho. Na Secção 2.3, é feita uma abordagem mais orientada à tecnologia e abordam-se ainda os modelos de programação no âmbito deste trabalho, procurando descrever a arquitetura de execução, as técnicas de otimização usualmente consideradas, as bibliotecas e as ferramentas de análise e *profiling* disponíveis.

No Capítulo 3, descreve-se o trabalho realizado: a solução atual e a forma como está implementada na Secção 3.1; a avaliação inicial realizada, na Secção 3.2; e, na Secção 3.3, a estratégia utilizada para, a partir dos dados obtidos na avaliação, otimizar a solução existente.

No Capítulo 4 faz-se a descrição das ferramentas e metodologias de avaliação e

validação utilizadas na Secção 4.1 e expõe-se a comparação dos ganhos obtidos com a implementação inicial, na Secção 4.2.

No Capítulo 5 fazemos, na Secção 5.1, o balanço do trabalho realizado, e terminamos na Secção 5.2, em que se apontam direções para trabalho futuro.

2

2 Trabalho Relacionado

2.1 Computação de Elevado Desempenho

A necessidade de resolver problemas complexos e imprimir maior rapidez na obtenção de respostas levou à evolução contínua da capacidade de processamento materializada inicialmente pelas máquinas de cálculo, passando pelas máquinas de processamento eletrónico, dos supercomputadores, dos computadores pessoais e atualmente em inúmeros dispositivos ubíquos com capacidades de processamento outrora impensáveis. O crescente desempenho dos supercomputadores (Figura 2.1) é consequência da evolução tecnológica conforme se pode comprovar, por comparação entre a calculadora síncrona paralela *Harvard Mark I* da IBM construída em 1944 e o supercomputador *Cray XK7*, líder da tabela dos *TOP 500 Supercomputer sites* em 2012. A miniaturização da tecnologia e a inclusão destas em dispositivos de processamento comprova ainda as previsões de Moore pela utilização de transístores *3-D tri-gate* de 22 *nm* [20].

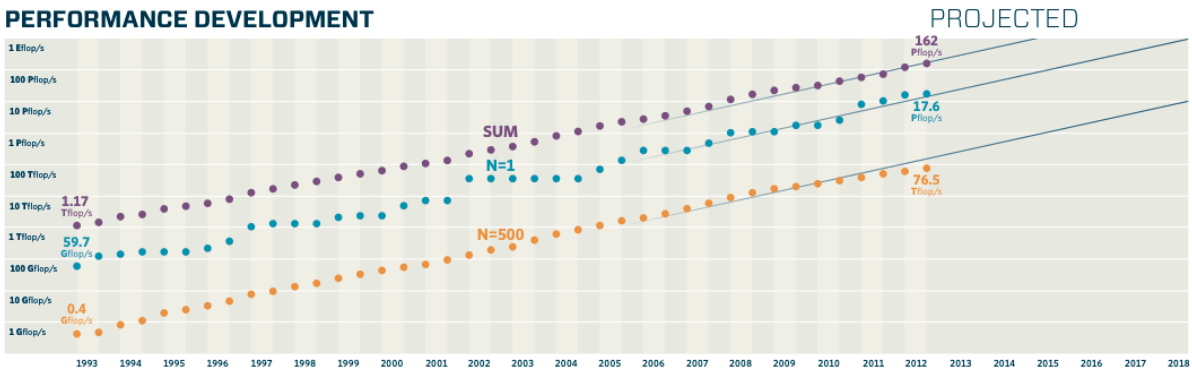


Figura 2.1: Evolução do desempenho [21]

O desenvolvimento contínuo de dispositivos eletrónicos de maior complexidade mas de maior desempenho, capaz de duplicar as suas capacidades de processamento, transferência de dados, armazenamento, entre outros, conduziu a um limite das possibilidades das arquiteturas existentes, obrigando ao desdobramento da tecnologia de forma que, com as mesmas capacidades, se consiga obter mais e melhores resultados. Barreiras físicas como a dissipação

do calor e interferências eletromagnéticas resultante da crescente densidade dos chips estão a conduzir a investigação para outras soluções, como a paralelização do processamento em dispositivos especializados até ao processamento quântico.

2.1.1 Arquiteturas

A arquitetura simples de *Von Neumann*, constituída por uma Unidade Central de Processamento (CPU), uma memória principal e um sistema de entrada e de saída de dados, com capacidade para executar sequencialmente conjuntos de instruções, teve desde as suas origens vários aperfeiçoamentos que se consubstanciaram atualmente em arquiteturas capazes de garantir bons desempenhos. As melhorias obtidas variaram desde a inclusão de novos dispositivos por *hardware* para auxiliar o processamento, maiores velocidades de transferência de dados, no número e especialização das instruções a executar, entre outras. Interessa neste trabalho referir aquelas que, mais diretamente contribuíram para o aumento da capacidade de processamento.

Um dos primeiros aperfeiçoamentos ao nível do processamento foi executar um número crescente de instruções complexas, ***Complex Instruction Set Computer (CISC),*** realizando várias tarefas por ciclo. Caracterizavam-se como instruções complexas e especializadas, de tamanho variável, com suporte a vários modos de endereçamento, número reduzido de registos, instruções até 2 operandos e suporte para operandos em memória. Apresentava a vantagem de criar programas com menos instruções transferindo o *overhead* para o *hardware* mas tinha como desvantagens a crescente complexidade das instruções e a sofisticação da unidade de controlo. Na década de 80, questiona-se a necessidade de tão grande conjunto de instruções quando apenas se utilizavam uma parte destas [22, p. 463].

A arquitetura ***Reduced Instruction Set Computer (RISC)*** surge da necessidade de reduzir a complexidade da arquitetura CISC, consequentemente simplificando o *hardware* e explorando o paralelismo ao nível das instruções. Assim, surgem instruções mais simples, de igual tamanho, capazes de serem executadas num único ciclo. Reduziu-se os modos de endereçamento disponíveis, aumentaram o número de registos de uso genérico e transferiu-se a complexidade das instruções para o compilador. Permitiu ainda utilizar ciclos de relógio mais curtos, aumentando o número de tarefas executadas e explorar de forma mais eficiente o *pipelining* das instruções [22, pp. 463-467].

A arquitetura ***Very Long Instruction Word (VLIW)*** procurou explorar maior paralelismo ao nível das instruções, recorrendo a instruções longas, de igual tamanho, compostas por múltiplas operações (lógicas, aritméticas e de controlo). O compilador possui um quadro geral das dependências das instruções e é responsável por dirimir as dependências aquando da compilação; contudo, em tempo de execução não consegue fixar o quadro de dependências e utiliza um *scheduling* conservativo. As instruções são agregadas em pacotes e são enviadas às várias unidades de execução. Uma extensão a esta arquitetura, a ***Explicity***

Parallel Instruction Computing (EPIC) conseguiu superar a construção de dependências do código em tempo de execução. O compilador é responsável pelo paralelismo das instruções combinando a especulação, predição e paralelismo explícito. [22, pp. 473-474]

As ***arquiteturas superescalares*** vieram dar a capacidade de execução simultânea de múltiplas instruções em cada ciclo de relógio, obtendo paralelismo através de *pipelining* e replicação. O conceito superescalar inclui *superpipelining*, *fetching* simultâneo de múltiplas instruções, uma unidade complexa de *decoding* (capaz de determinar a dependência das instruções e combinar as instruções dinamicamente de forma a que não ocorra violação de dependências) e um conjunto de recursos para a execução paralela de múltiplas instruções. Possui várias unidades de execução (vários somadores e multiplicadores de inteiros e virgula flutuante bem como outros componentes em *hardware* especializados) que podem trabalhar de forma independente [22, p. 473].

A mais conhecida forma de categorizar as arquiteturas de computadores, a ***taxonomia de Flynn***, foi proposta por *Michael Flynn* em 1966, tendo como base a seguinte combinação: fluxos simples/múltiplos de instruções/dados [23] (Figura 2.2).

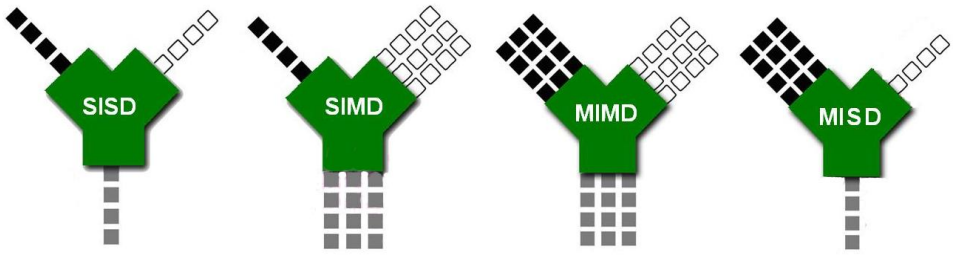


Figura 2.2: SISD – Taxonomia de Flynn (Imagem adaptada) [24]

Single Instruction Stream, Single Data Stream (SISD), corresponde à máquina sequencial de *Von Neummann*. Enquadram-se nesta categoria os computadores com processador único, sem paralelismo, onde uma instrução é descodificada numa unidade de tempo e o fluxo de instruções opera sobre um fluxo de dados.

Single Instruction Stream, Multiple Data Stream (SIMD) possui uma unidade de controlo, uma memória para instruções e memórias para os dados, executando a mesma instrução simultaneamente sobre múltiplos dados. Permite o paralelismo e enquadram-se nesta categoria os ***processadores vetoriais*** e os ***processadores matriciais***. Os processadores vetoriais são altamente *pipelined* e especializados que realizam a mesma operação em paralelo sobre todos os elementos de um vetor enquanto que, nos processadores matriciais, a execução das instruções é realizada por um conjunto de processadores mas é controlada por uma unidade de controlo. Os processadores vetoriais mais bem-sucedidos possuem um conjunto de registos especializados que podem conter vetores inteiros e que, com uma única operação deslocam os dados da memória para os registos e vice-versa. Cada instrução equivale a um ciclo e não existe dependência de dados. Estão divididos em 2 categorias,

definidas pela forma como as instruções acedem aos operandos: Memória-Memória e Registador-Registador. As instruções vetoriais são eficientes porque o número de *fetchs* é menor, o que reduz o número de operações de descodificação e, consequentemente, o *overhead* sobre a unidade de controlo; segue-se a menor largura de banda utilizada e, sabendo que se tem uma sequência contígua de dados, pode fazer-se o *prefetching* dos correspondentes pares de valores. Tem como desvantagens o facto de que, para vetores de tamanho superior aos registos é necessário dividir os dados em segmentos. As arquiteturas SIMD são cada vez mais usadas em aplicações para o cálculo científico, processamento de matrizes, análise de imagem, codificação de áudio e vídeo e reconhecimentos de padrões [22, pp. 474-475].

Multiple Instruction Stream, Single Data Stream (MISD) baseia-se numa cadeia de processadores no qual os dados passam de processador em processador, onde sofrem modificações. Não se conhecem máquinas comerciais com esta arquitetura sendo que, caso existam, estas se destinam a finalidades específicas (múltiplos filtros de frequência a operar sobre um fluxo de sinal, algoritmos de criptografia para cifrar/decifrar mensagens, múltiplas divisões para verificar se um número é primo) e não a uso geral [22, pp. 467-471].

Multiple Instruction Stream, Mutiple Data Stream (MIMD) no qual um conjunto de processadores executa simultaneamente sequências diferentes de instruções sobre diferentes conjuntos de dados. Esta arquitetura ainda se pode subdividir quanto à forma como os processadores comunicam entre si: memória partilhada ou por mensagens. Os três tipos de arquiteturas MIMD mais utilizadas são S*ymmetric Multiprocessor (SMP), Nonuniform Memory Acess (NUMA) e Cluster.*

Na comunicação por memória partilhada, existe uma memória global partilhada pelo qual os processos comunicam, coordenam e sincronizam-se. Tem como desvantagens o facto de possuir barramento único à memória (que no caso de existirem muitos processadores a aceder à mesma resulta num *bootleneck*) e ainda de ser pouco escalável. Soluções para ultrapassar estas limitações passam por fazer a replicação dos dados em caches locais, mas isso conduz a problemas de consistência e coerência de dados; recorre-se então a *hardware* especializado, o que aumenta a complexidade dos sistemas [22].

Nonuniform Memory Acess (NUMA) cada processador tem na sua proximidade física um espaço de memória, sendo que o espaço de endereçamento é único para todos os processadores. O tempo de acesso à memória dependerá portanto da proximidade a que estiver o endereço pesquisado. Para reduzir os tempos de acesso, os processadores dispõem de caches mas estas introduzem de novo as questões da coerência e consistência dos dados, resolvidas com a utilização de *snoopy cache controllers* (arquiteturas CC-NUMA), recurso a políticas de atualização *write-through* (*update* ou *invalidation*) ou *write-back* [22, pp. 483-484].

Symmetric Multiprocessor (SMP) possuem dois ou mais processadores similares,

partilham uma mesma memória e acessos aos mesmos dispositivos de entrada e saída de dados, sob um sistema operativo comum, desempenhando as mesmas funções. O tempo de acesso à memória é igual para todos os processadores e partilham igual tempo de acesso ao *bus*. Contudo, o *bus* constitui-se um *bootleneck* quando vários processadores pretendem aceder simultaneamente à memória [1, pp. 632-633].

A ***arquitetura MIMD com memória distribuída*** é constituída por nós, compostos por unidades de processamento e memória e que comunicam entre si através de mensagens numa rede. O exemplo mais conhecido desta arquitetura é o ***cluster***, que tem como vantagens ser escalável e possuir um bom desempenho mas é penalizada pelo facto da troca de mensagens tornar mais complexa a tarefa de coordenar as comunicações. Existem dois tipos de *clusters*: de alto desempenho (grande capacidade de processamento para execução de tarefas intensivas) e de elevada disponibilidade (replicação de nós de forma a eliminar pontos de falha). Existem ainda arquiteturas de processamento distribuído, que consiste em redes de computadores que trabalham de forma colaborativa na resolução de problemas complexos, tendo como exemplos: a computação em *Grid* e aglomerados BEOWULF dentro de redes locais. O processamento distribuído global, uma extensão à computação em *Grid* tendo em conta que o aproveitamento dos recursos computacionais extravasa as redes locais, utilizam redes globais como a internet e servem para resolver problemas de enorme complexidade. Exemplos desses projetos são por exemplo, o SETI@Home [22, p. 485].

Já fora da taxonomia de *Flynn* e como resposta à necessidade de encontrar novas formas de processamento de maior desempenho, a investigação avança para arquiteturas alternativas tais como, *dataflow computing*, redes neuronais e processamento quântico [22, pp. 487-496]

2.1.2 Modelos de programação

Numa perspetiva abstrata, os dois paradigmas-base de interação são a partilha de memória e a troca de mensagens também designados, respetivamente, modelos de espaço de endereçamento partilhado e distribuído. Tais paradigmas são oferecidos ao programador por linguagens e/ou bibliotecas de programação, sendo que os programas neles realizados são depois executados sobre sistemas físicos (i.e., computadores) cujas arquiteturas são, elas próprias, de memória partilhada (e.g., multiprocessadores SMP ou ccNUMA) ou de memória distribuída (e.g., nós numa rede).

Sendo a gestão da memória uma das principais funções de um sistema de operação é natural que deste seja também a responsabilidade da sua partilha; os SO modernos oferecem ao programador APIs (e.g., *System V IPC*) para partilha de memória entre processos distintos; contudo, atualmente, a forma mais "natural" de partilhar memória entre múltiplas entidades ativas recorre a *threads* que se executam no interior de um processo, tal como realizada em linguagens (e.g., C) que permitem a programação com bibliotecas que suportam a norma *Pthreads*, ou outras (e.g., Java) nas quais as *threads* são nativamente oferecidas. Do mesmo

modo, i.e., nativamente ou sob a forma de bibliotecas, são oferecidas ao programador APIs de troca de mensagens, sendo exemplos a API de *sockets* (TCP/IP, por exemplo) ou a MPI – *Message Passing Interface*.

2.1.3 Curvas de aprendizagem e produtividade

Da mesma forma que não há uma linguagem "universal" que permita programar, com o mesmo grau de facilidade (ou dificuldade), todas as diferentes tarefas que se apresentam, o mesmo se passa com os modelos de interação: casos há em que a solução se torna, de um ponto de vista conceptual, mais simples se considerarmos o paradigma de memória partilhada do que se adotarmos o da troca de mensagens. Um exemplo que ilustra muito bem esta questão é a multiplicação de matrizes: é muito simples, para quem está habituado a desenvolver programas sequenciais, mostrar como com pequenas alterações no algoritmo, se pode acelerar a multiplicação usando múltiplas *threads* que se executam num SMP; contudo, o oposto acontece quando se mostra como resolver o mesmo problema – para se acelerar a execução – usando trocas de mensagens.

É, portanto, comum usar-se o termo barreira semântica (*semantic gap*) para referir as dificuldades inerentes à programação concorrente, ou paralela, quando realizada com um ou outro paradigma; no caso da memória partilhada o programador tem de compreender os aspetos da sincronização explícita usando mecanismos como semáforos, mutexes, barreiras, etc., enquanto a utilização do modelo de troca de mensagens obriga não só a repensar os algoritmos como, quando o objetivo é acelerar a execução, considerar questões como o tempo de troca de mensagens – tanto na perspetiva da latência como na da largura de banda da comunicação.

Assim, o desenvolvimento de aplicações concorrentes ou paralelas requer uma longa aprendizagem e, mesmo assim, tem, quando comparada com o desenvolvimento de aplicações sequenciais, uma relativa baixa produtividade.

Vários passos têm sido dados no sentido de "aliviar" ou remover alguns dos obstáctulos que são responsáveis por esta "baixa produtividade"; um exemplo recente é o *OpenMP*[1], que utiliza paralelismo explícito com recurso a diretivas de compilação, bibliotecas de funções e variáveis de ambiente. As diretivas fornecem suporte para a concorrência, sincronização e gestão de dados, tentando evitar o recurso explícito a mecanismos de sincronização. As diretivas são baseadas em pragmas que permitem explicitar ao compilador o paralelismo, e controlá-lo. Um modelo possível é o *fork-join*, no qual o programa é executado sequencialmente até encontrar a diretiva *parallel*, altura em que a *thread* se transforma numa

1 A Open Multiprogramming Plataform (OpenMP) é uma plataforma suportada por vários fabricantes de software que especificaram uma API para programação paralela em arquiteturas de multiprocessadores para ser utilizada em C/C++ e Fortran.

thread master e são criadas um certo número de novas *threads*, número esse que pode ser definido explicitamente pelo utilizador. As *threads* possuem variáveis locais e/ou partilhadas (previamente inicializadas ou criadas no ponto onde são criados as *threads*) e cláusulas de redução que especificam como são combinadas as múltiplas cópias locais de uma variável numa única cópia quando as *threads* terminam. Este modelo de programação tem como vantagem a facilidade de conversão de programas sequenciais em programas paralelos e a simplicidade de utilização via diretivas [25] [26].

2.1.4 Expectativas de desempenho

O foco deste trabalho é o desempenho assim, estamos fundamentalmente interessados em aumentá-lo recorrendo ao paralelismo, i.e., à utilização de múltiplas unidades executoras (ao nível macro podemos falar de nós, CPUs e *cores*, e a um nível mais elementar referir múltiplas ALUs e FPUs) que suportam a execução de múltiplas *threads* partilhando um mesmo espaço de endereçamento, ou múltiplos processos que trocam mensagens, ou ainda, numa realização híbrida, usando nós de computação interligados por uma rede na qual cada nó corre um processo *multithreaded* e troca mensagens com outros processos que correm noutros nós.

Numa perspetiva *naïve*, esperamos que acrescentando mais recursos a uma infraestrutura computacional, daí resulte uma melhoria no desempenho (*speedup*) da aplicação; por exemplo, podemos esperar (desejar) que um aumento na velocidade de relógio da CPU se traduza por uma igual diminuição no tempo de execução da aplicação. Contudo, já sabemos que nem sempre – ou melhor, muitas das vezes – tal acontece, mesmo em "simples" programas sequenciais.

Assim, no caso das aplicações paralelas, a complexidade das questões relacionadas com o *speedup* das aplicações resulta ainda maior do que no caso das aplicações sequenciais, pois agora há que considerar as situações em que, mesmo que um recurso – e.g., CPU – esteja livre não pode ser usado porque, por exemplo, o processo aguarda pela chegada de uma mensagem, ou a *thread* não pode aceder à informação pois o *mutex* que a protege está trancado (*locked*).

2.2 Computação com GP-GPUs

Uma *Graphics Processing Unit* (GPU) é, como o próprio nome indica, uma unidade capaz de processar autonomamente, i.e., sem intervenção da CPU, objetos gráficos tais como linhas, arcos, retângulos e caracteres. O contínuo desenvolvimento, motivado pelas aplicações gráficas *realtime* e 3D, do *pipeline* gráfico nas últimas três décadas traduziu-se num enorme acréscimo da capacidade de processamento das GPUs, que apresentam um cada vez melhor desempenho e menor custo.

Os dispositivos usados em computação para potenciar a capacidade de processamento são designados por aceleradores; são exemplos de aceleradores os *Application-Specific*

Integrated Circuits (ASIC) [27], *Field Programmable Gate Arrays (FPGA)* [28] [29], *Cell Broadband Engines (Cell BE)* [30] e G*eneral Purpose Graphics Processing Units (GP-GPU)*. Os dois primeiros são "customizados" para aplicações específicas (por exemplo, criptografia, ou geração/comparação de padrões de redundância para discos RAID); os dois últimos são programáveis, embora especialmente vocacionados para aplicações que processam quantidades maciças de dados com padrões regulares (e.g., matrizes), incluindo-se aqui as aplicações gráficas.

2.2.1 Evolução das Arquiteturas

A evolução da arquitetura das GPUs está diretamente ligada ao contínuo desenvolvimento dos recursos associados ao processamento gráfico: a primeira fase é caracterizada pelos **dispositivos de rasterização** que tinham apenas a função de mostrar os *pixéis* desenhados no ecrã, com todo o processamento gráfico feito pela CPU. Contudo, com a necessidade de visualizar mais rapidamente cenas mais complexas sem monopolizar a CPU, transferiu-se para a GPU a capacidade de processamento das geometrias, tais como: o controlo dos vértices, o *vertex shading*, *transform and lighting (VS/T&L)*, construção de triângulos, a rasterização, sombreamento, operação de rasterização final (cor, transparência, *anti-aliasing*, objetos ocultos), como também a gestão dos *buffers* de memória para leitura e escrita no dispositivo de visualização. Esta é designada como a fase da ***Fixed Function GPU*** no qual cada estágio é configurável, mas não é programável [31, pp. 23-24]. Os cálculos realizados serviam apenas para o processamento gráfico e não é possível aceder ao processador para o utilizar para outras tarefas. Surgem nesta altura APIs que simplificam a comunicação com o processador gráfico, sendo as mais populares o *DirectX* da *Microsoft* e o *standard* aberto *OpenGL*. A *NVIDIA* populariza nesta altura o termo GPU ao apresentar uma arquitetura com capacidade para gerar 15M polígonos/segundo e um desempenho de 480M pixels/segundo [32]. A partir de 2001, surge a fase dos **GPUs programáveis** [33, p. 5] que permitem o acesso programático às funcionalidades da geometria *Vertex and Pixel Shaders*, e à memória das texturas; aparecem as placas *NVIDIA GeForce 3* (em 2001) e *ATI Radeon 9700* (em 2002) e as *APIs DirectX* 8 e 9 e *OpenGL vertex shader extensions*.

A **arquitetura unificada** das GPUs surge em 2006, quando a *NVIDIA* mapeia os estágios programáveis do *pipeline* num *array* unificado de processadores com vários *fixed functions* GPUs, no qual a lógica sequencial do *pipeline* gráfico é vertida num circuito fechado, onde os dados passam ciclicamente pelos processadores *(*Figura 2.3): os dados são recebidos do *host* sobre a forma de vértices,

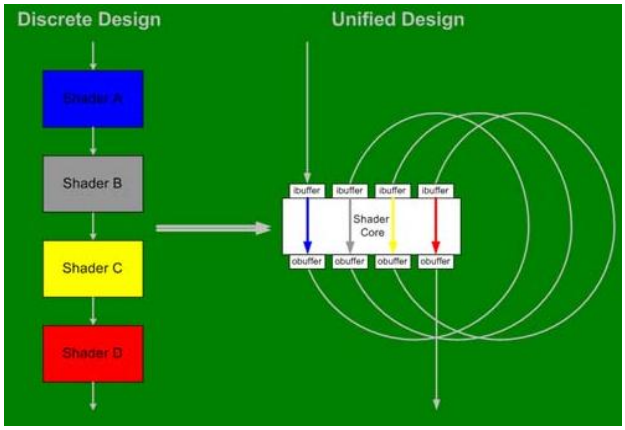


Figura 2.3: Pipeline clássico e arquitetura unificada [34]

são processados pelo *shader core*, para onde os resultados são novamente enviados até que estejam prontos a serem enviados para o sistema de rasterização.

Nesta arquitetura, designada NVIDIA G80 e apresentada com maior detalhe na Figura 2.4, os *unified stream processors* (SP) podem processar vértices, pixéis, geometrias e cálculos físicos, equiparando-se a unidades de processamento de vírgula flutuante, de âmbito geral, independentes entre si, capazes de realizar as operações aritméticas elementares. O *dispatch* e a unidade de controlo da GPU podem atribuir dinamicamente vértices, geometrias ou operações de pixéis aos SPs sem quaisquer condicionalismos ou restrições. Com esta arquitetura potencia-se o balanceamento de carga dos SP e obtém-se maior eficiência dos recursos disponíveis.

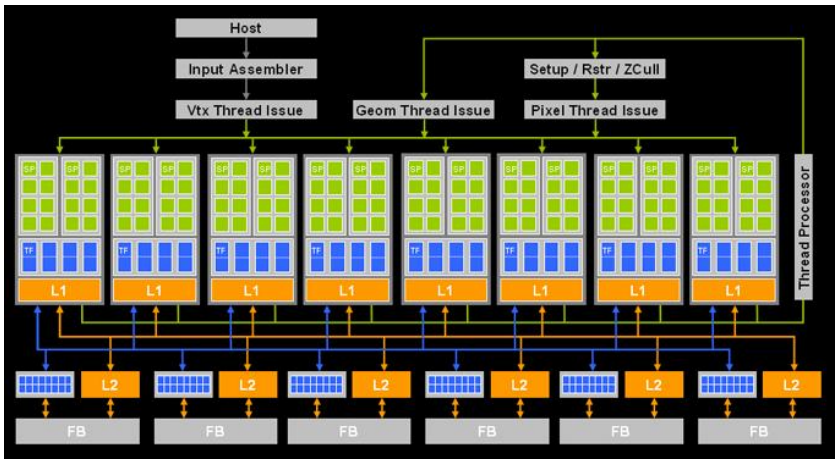


Figura 2.4: Arquitetura geral Arquitetura geral da GeForce 8800 GTX [34]

O sucesso desta arquitetura deve-se ao agrupamento e utilização eficiente de numerosos SP escalares que executam operações sobre fluxos de entrada de dados, produzindo outros fluxos que alimentam outros SP. Os SP possuem instruções de descodificação altamente especializadas e rápidas e uma lógica de execução que opera sobre os diferentes fluxos de dados que podem ser armazenados na memória do dispositivo (caches L1/L2 e memória de texturas) (Figura 2.5).

Atendendo à forma como os dados são guardados, em vetores, esta arquitetura utiliza instruções vetoriais e matriciais de forma a poder obter maior eficiência. Enquadra-se portanto numa arquitetura do tipo SIMD, que permite ainda constituir *clusters* de SPs [34].

De forma genérica, esta GPU foi concebida para o processamento intensivo, altamente paralelizável, exatamente o que era necessário para o processamento e renderização gráfica. Contudo, vários investigadores souberam aproveitar habilmente estas capacidades para resolverem problemas noutras áreas da ciência. Para tal, foi necessário recorrer a adaptações das estruturas de dados utilizadas, no seu mapeamento em memória, na lógica de programação e execução dos programas, conforme as especificidades da tecnologia em uso, abrindo caminho para o conceito de *General Purpose GPU*.

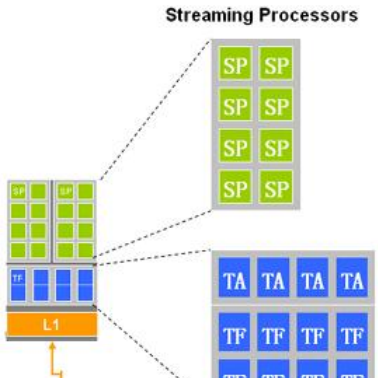


Figura 2.5: Organização do SP e memória

Nos últimos 25 anos duas grandes empresas se sobressaíram, a *NVIDIA* e a *Advanced Micro Devices (AMD)*, que concorreram para o rápido desenvolvimento das capacidades de processamento das GPUs e na disseminação destes dispositivos por um vasto leque de tecnologias, tais como: computadores pessoais, dispositivos móveis, centros de dados e sistemas integrados [35] [36]. Importa então aqui descrever nas secções seguintes a evolução das arquiteturas desenvolvidas pelas duas empresas, fazer um ponto de situação atual e comparar ambas.

Arquiteturas *NVIDIA*

A primeira realização de uma arquitetura GP-GPUs que se conhece é da NVIDIA, em novembro de 2006, com a linha de placas gráficas GForce 8800 e foi designada por **arquitetura G80** (Figura 2.4) contendo 8 *Texture/Processing clusters* (TPC) com 2 *streaming multiprocessor* (SM) contendo cada um destes 8 *stream processor* (SP) num total de 128 SP. Ela trouxe inovações tais como: unificou os *pipelines* dos vértices e pixéis num processador único, capaz de fazer processamento geral com algumas adaptações; utilizou um processador escalar de *threads* eliminando a necessidade de programar manualmente os vetores de registos; trouxe um novo modelo de execução *Single Instruction Multiple Thread* (SIMT) onde múltiplas *threads* independentes executam a mesma instrução; apresentou os mecanismos de comunicação inter-*threads* baseados em memória partilhada e barreiras de sincronização; e apoiou o desenvolvimento da programação para GPUs ampliando a linguagem C [37].

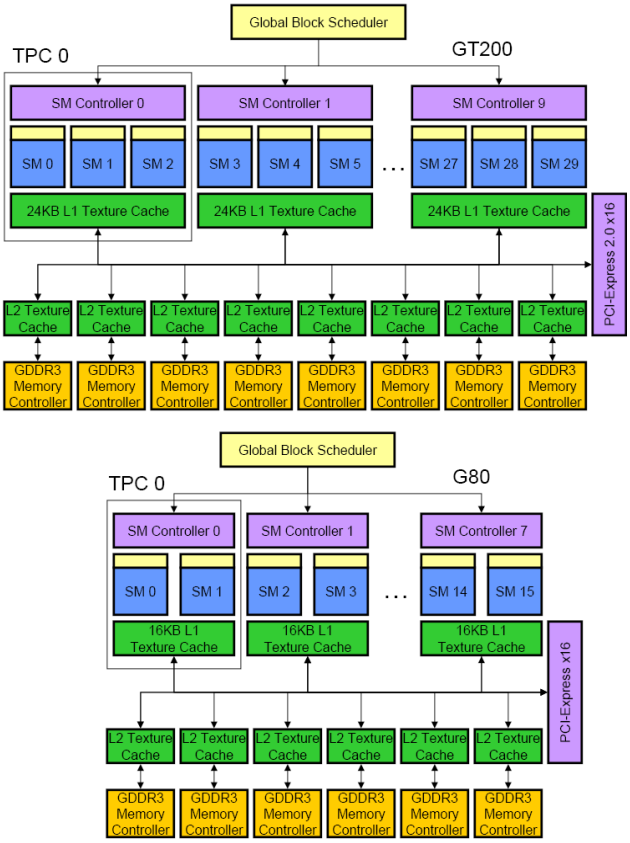


Figura 2.6: Arquitetura G80 & G200 [75]

A arquitetura G80 deu origem à **arquitetura GT200** que equipou as placas *GeForce* GTX 280, Quadro FX 5800 e GP-GPUs Tesla T10. Esta nova arquitetura incrementa o número de SP para 240, duplica o número de registos (aumentando o número de *threads* a executar num determinado momento), permite coalescer os acessos à memória e adiciona suporte a operações de vírgula flutuante de dupla precisão a 64 bits.

Não havendo grandes alterações de arquitetura relativamente à anterior (Figura 2.6), descrevemos sumariamente os seus componentes: na arquitetura GT200 existem 10 TPC cada um com 3 SM e uma *cache* L1 de texturas de 24KB partilhada por todos os SM; cada SM possui 8 SP, cada um com três unidades aritméticas: de vírgula flutuante, de inteiros e comparadora. Possui ainda um *bus* de ligação à *cache* L2 das texturas e à interface PCI-Express X16.

Um SM (Figura 2.7) é constituído por uma *cache* de instruções, um controlador, um escalonador, registos, *cache* para constantes, memória partilhada, comparadores/somadores (inteiros e virgula flutuante) duas unidades de funções especiais (para funções transcendentes, trigonométricas, etc.) e somadores/multiplicadores.

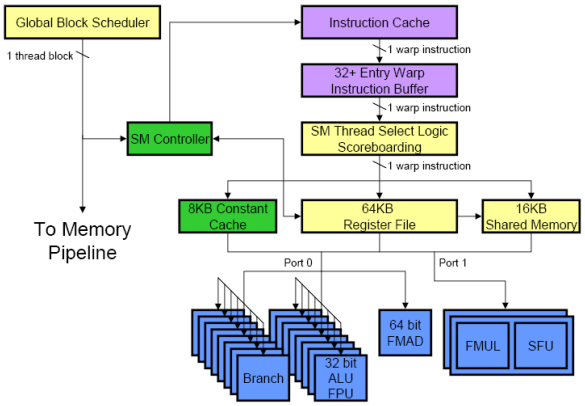


Figura 2.7: Organização logica de um SM da arquitetura GT200 [75]

Para responder a novos requisitos e aperfeiçoar as arquiteturas anteriores, a NVIDIA introduz em 2010 uma nova arquitetura, a **arquitetura Fermi:** dispondo de 4 *Graphics Processing Clusters* (GPCs) cada um contendo 4 SM, num total de 16 SM, e os SM passam a dispor de 32 CUDA *cores* (outra designação para os SP). O desempenho das operações de vírgula flutuante é melhorado, o escalonador agenda e despacha instruções simultaneamente, são acrescentados 64KB de RAM configuráveis como memória partilhada ou *cache* L1 (48KB memória partilhada e 16KB de *cache* L1 ou o inverso) e unificação da *cache* L2. Cada CUDA *core* executa uma instrução inteira ou de vírgula flutuante por ciclo de relógio e por *thread*. A GPU suporta um total de 6GB de memória GDDR5 DRAM e possui um interface PCI-*Express* que liga o *host* à GPU; a memória está subdividida em 6 módulos, acessíveis em paralelo por um bus de 6x64 bits. Possui uma unidade *GigaTread* que efetua o agendamento global e a distribuição dos blocos de *threads* pelos *threads schedulers* dos SM. Tudo isto não só aumenta o desempenho, como simplifica a programação e torna-a mais eficiente [37].

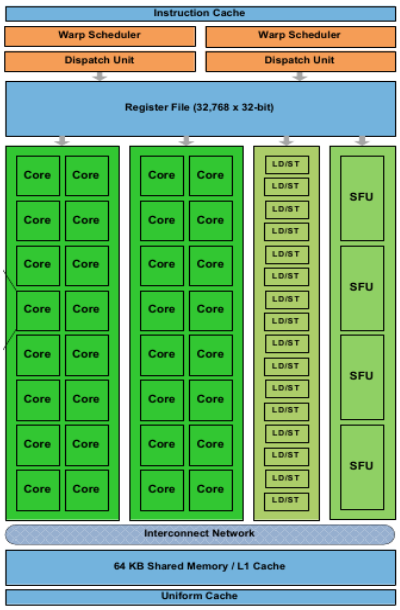


Figura 2.8: Arquitetura do SM [37]

Cada SM (Figura 2.8) possui então 32 CUDA *cores*, 16 unidades de *load*/*store* (para carregamento de dados, que serão operados por um bloco de 16 *threads*), 4 *Special Function Units* (SFU), para execução de funções transcendentes, uma *cache* de instruções, 2 unidades

paralelas de agendamento e despacho de instruções, um ficheiro de registos, 64KB de memória partilhada e *cache* L1 configurável. Cada SFU executa uma instrução por *thread*, por ciclo de relógio. Cada CUDA *core* é constituído por dois *pipelines* um para cálculo aritmético de inteiros e outro para vírgula flutuante (Figura 2.9).

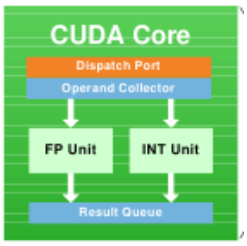


Figura 2.9: CUDA core [37]

Uma das inovações da arquitetura, e que trouxe uma melhoria ao nível da programação e do desempenho, foi a utilização de memória partilhada no próprio dipositivo, permitindo a cooperação de *threads* dentro de um mesmo bloco, a reutilização dos dados e a redução das transferências de dados para fora do dispositivo. Para responder a diferentes problemas e utilizações, as memórias e caches foram hierarquizadas (Figura 2.10) de forma que a memória partilhada e *cache* L1 ficam acessíveis aos blocos dentro de um SM, e a *cache* L2 foi unificada de forma a responder a todas as solicitações (*load*/*store* e texturas).

Outras melhorias obtidas foram: suporte ao novo conjunto de instruções *Parallel Thread eXecution* (PTX) [2]; espaço de endereçamento unificado relativamente à memória local da *thread*, à memória partilhada do bloco e memória global para operações de *load* e *store* (Figura 2.11), otimização do suporte ao *OpenCL* [3] e *DirectCompute*[4], suporte a normas IEEE 32-bits *Floating Point Precision*, suporte ao *Error Correcting Code* (ECC) para dados em memória, execução de *kernels* concorrentes. Ocorreram também melhorias no desempenho, nomeadamente: da execução condicional (através da predição), de operações atómicas em memória, e de mudanças de contexto das aplicações, etc.

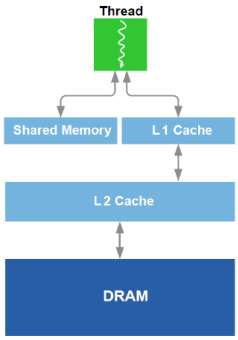


Figura 2.10: Hierarquização da memória

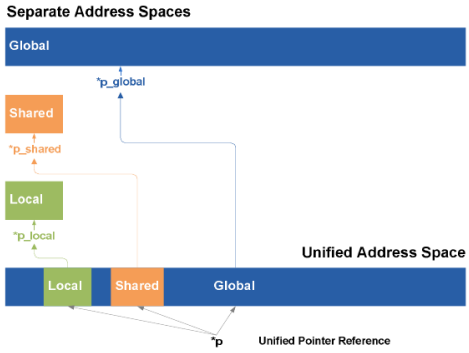


Figura 2.11: Espaço de endereçamento unificado.

2 Máquina virtual ISA de baixo nível desenhada para suportar operações de um processador de threads paralelos. Quando ocorre a instalação do programa as instruções PTX são traduzidas para instruções máquinas pelo driver do GPU.

3 É um padrão aberto para programação paralela de propósito geral, em GPUs, CPUs e outros dispositivos de processamento, oferecendo portabilidade e eficiência.

4 API que tira proveito do poder de processamento paralelo da GPU para acelerar o desempenho dos sistemas operativos Windows Vista e do Windows 7

Tendo em consideração a crescente utilização de GPUs no processamento de alto desempenho em várias áreas da ciência tais como medicina, finanças, e simulações de física e bioquímica, entre outras, a NVIDIA procurou otimizar e obter maiores rendimentos do processamento paralelo, simplificando a criação de programas paralelos com a apresentação de uma nova arquitetura designada como **Kepler**. A nova arquitetura (Figura 2.12), apresentada em 2012 [38], tem as seguintes características: mantém a estrutura base dos 4 GPC como unidade base, define uma nova arquitetura do SM agora designado SMX; um subsistema de memória que oferece novas capacidades de *cache*; maior largura de banda a cada nível da arquitetura e uma implementação I/O DRAM mais rápida; e suporte por *hardware* a novas funcionalidades do modelo de programação.

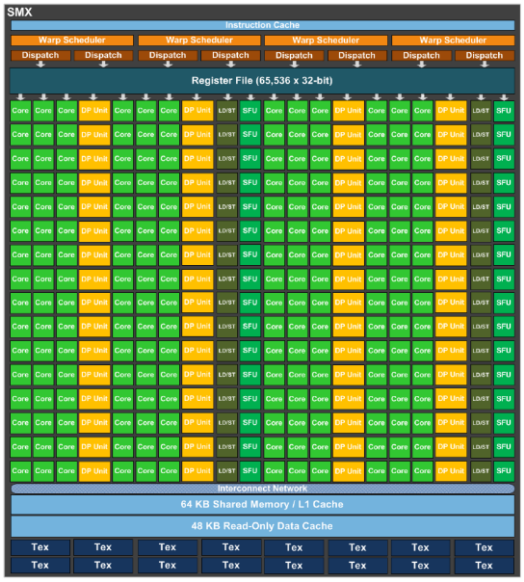


Figura 2.12: Arquitetura Kepler

Comparativamente à arquitetura anterior, as grandes alterações decorrem ao nível dos SMX que passam a ter 192 CUDA *cores* de precisão simples, 64 unidades de dupla precisão, 32 unidades de funções especiais e 32 unidades de *load*/*store*, e 4 *warp schedulers* (cada um deles com dois *dispatchers*). O facto de possuir 4 *schedulers* e 8 dispatchers permite que concorrentemente se possa ter 4 *warps*[5] em execução com duas instruções diferentes por *warp*, por cada ciclo de relógio.

Foram ainda introduzidas outras otimizações de desempenho, tais como: 255 registos por *thread*; partilha de dados entre *threads* sem que haja necessidade de *load*/*stores*; operações atómicas em memória global e estendida a 64 bits e aumento de número de filtros de textura e gestão de objetos de textura em memória.

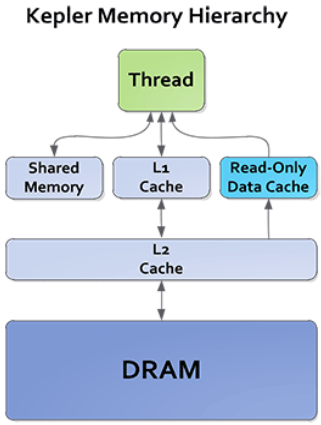


Figura 2.13:Hierarquia da memória Kepler

Relativamente ao subsistema da memória ele é similar ao da Fermi, mas com algumas extensões: permite ao compilador o acesso a uma *cache* de leitura de 48KB (Figura 2.13); maior flexibilidade de configuração da memória partilhada e *cache* L1, permitindo criar partições de 32KB para cada uma; a dimensão da *cache* L2 passa para 1536KB; o suporte ECC não só é aplicado à

[5] Na arquitetura SIMT da NVIDIA, um *warp* é o conjunto de 32 *threads*.

memória, como também aos registos e caches.

Foram ainda adicionadas novas funcionalidades, tais como:

- Paralelismo dinâmico no qual a GPU gera novos fluxos internos de processamento, faz a gestão e controlo desses fluxos e sincroniza os resultados no final sem ação da CPU. Assim, um *kernel* pode gerar outro *kernel*, pode criar *streams*, eventos e gerir dependências. Liberta a CPU destas tarefas e fornece a possibilidade ao programador de realizar tarefas recursivas, explorar a dependência dos dados, e executar mais tarefas na GPU.
- Hyper-Q, que incrementa o número de ligações ou filas de trabalho entre o *host* e o distribuidor de tarefas de CUDA, possibilitando 32 ligações simultâneas geridas por *hardware*.
- Unidade da Gestão da *Grid*, que pretende manter a GPU sempre em utilização e de forma eficiente. Com a utilização da funcionalidade do paralelismo dinâmico, são gerados novos agrupamentos de blocos de *threads* e consequentemente novas *grids* que passam a ser geridas por esta nova unidade. Esta unidade, além de gerir, prioriza as *grids* e passa ao distribuidor de tarefas que as distribui pelos SMX.
- *GPUDirect* RDMA é uma funcionalidade de DMA remoto que permite a um GPU num *host* transferir dados de/para a memória de um GPU noutro *host* usando as NICs como dispositivos de transporte.

No presente ano a NVIDIA apresentou uma nova arquitetura, denominada por **arquitetura *Maxwel*** (1º geração) voltada para a eficiência energética e obtendo melhores desempenhos dos SM relativamente aos anteriores, 35% por core e duas vezes mais por unidade de *Watt* [39] [40].

A nível macro mantém a organização por GPCs contudo a organização dos SM foi alterada passando a designar-se por *streaming multiprocessor Maxwell (*SMM). A divisão dos recursos destes SMM em 4 subunidades com as respetivos SP, registos, *Instruction buffer*, *warp scheduler* e *2 dispatch units*, com algumas otimizações nos algoritmos de *scheduling*, no maior número de instruções despachadas por ciclo de relógio, na redução da

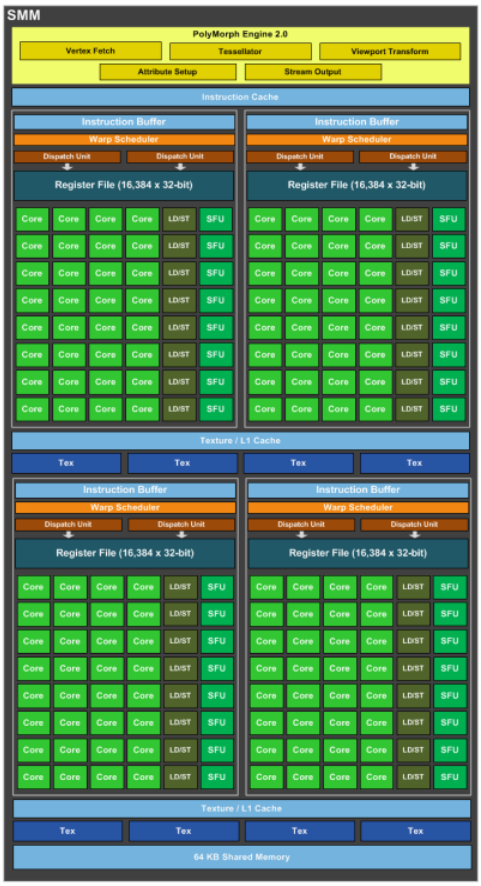


Figura 2.14: Multiprocessador Maxwell

latência da execução de instruções aritméticas, veio trazer melhor desempenho e eficiência destas unidades. Ainda de salientar nesta arquitetura: a atribuição de 64KB de memória dedicada a cada SMM, memória nativa partilhada para operações atómicas de 32 *bits* e memória nativa partilhada para operações de comparação e *swap* de 32/64 *bits.*

Em resumo, a NVIDIA procurou ao longo dos tempos aperfeiçoar a arquitetura que desenvolveu inicialmente (Tabela 2.1), numa primeira fase incluindo um conjunto de recursos adicionais que assim permitiu obter bons desempenhos mas com a penalização dos consumos energéticos e posteriormente, considerando esses custos, apostando em desenvolver tecnologias mais eficientes e modulares.

Tabela 2.1: Comparativo entre as diferentes arquiteturas da NVIDIA

GPU	G80	GT200 (Tesla)	GF110 (Fermi)	GK107 (Keppler)	GM107(Maxwell)
Transistors	681 *milion*	1.4 *bilion*	3.0 *bilion*	1.3 *bilion*	1.87 *bilion*
CUDA Cores /SM	8	16	32	192	64
SM /TPC or GPC	2	3	4	8	10
TPC or GPC	8	10	4	1+	1+
Graphics Core Clock	500MHz	648MHz	772MHz	1058MHz	1020MHz
Shader Core Clock	1350MHz	1476MHz	1544MHz	n/a	1085MHz
GFLOPs	-	1063	1581	812.5	1305.6
Memory Clock	1800MHz	2484MHz	4008MHz	5000MHz	5400MHz
Memory Bandwidth	86.4GB/sec	159GB/sec	192.4GB/sec	80GB/sec	86.4GB/sec
L1 / Shared Memory	16KB (SM)	16KB (SM)	16/48KB	16/32/48 KB	64KB
L2 Cache Size	-	-	768KB	256KB	2048KB
Register File Size / SM	-	-	128KB	256 KB	256 KB
TDP	135W	183W	244W	64W	60W
Manufacturing Process	90-nm	65-nm	40-nm	28-nm	28-nm

Arquitetura das GPUs/AMD

Desde cedo a AMD (*Advanced Micro Devices*) esteve envolvida no desenvolvimento de dispositivos computacionais, tanto CPUs como GPUs[6]. Uma vez que o foco deste trabalho é a otimização de uma aplicação CUDA, executável apenas sobre arquiteturas NVIDIA, a inclusão das arquiteturas AMD faz-se apenas por razões de completude, e será necessariamente muito breve; fosse o foco uma aplicação *OpenCL* e esta secção teria uma extensão idêntica, ou até superior à anterior.

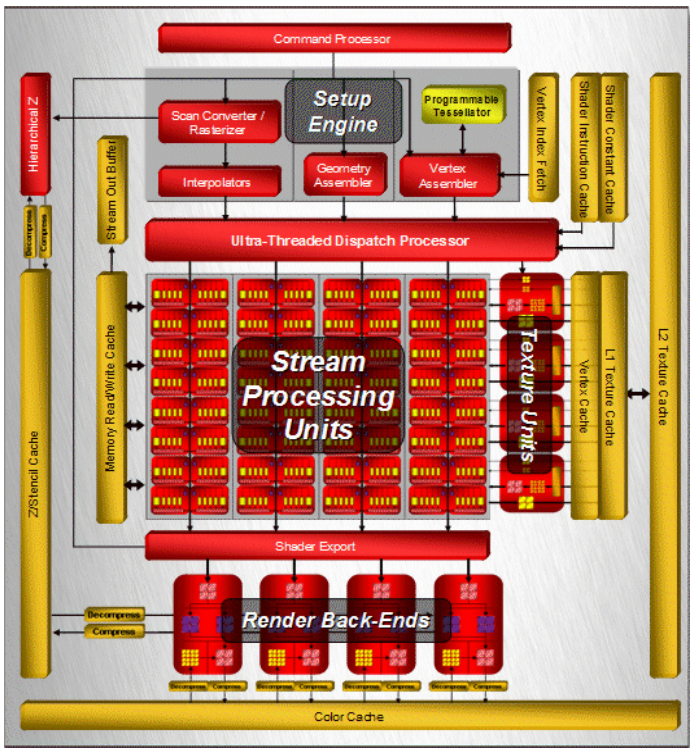


Figura 2.15: Arquitetura R600 [69]

A arquitetura R600, apresentada pela AMD em 2007 (Figura 2.15) é da mesma classe que da NVIDIA G80, e tem um desempenho semelhante. Diferencia-se por possuir uma unidade (designada *Tesselator*) para processar malhas, pela complexidade da sua programação (forma a obter maior rendimento das instruções muito longas (VLIW)); ser compatível com o *DirectX* 10 e apresentar um *shader* unificado no qual pode processar os dados referentes aos vértices, geometrias e pixéis do *pipeline* gráfico.

A unidade de processamento tem 320 *stream processors*, um *Setup Engine*, *render back-ends*, e outros dispositivos e é baseada em VLIW procurando extrair maior paralelismo da instrução. O *Setup Engine* é o responsável por preparar os dados para processamento e possui uma unidade *Vertex Assembler* para processar os vértices (e *Tesselator* para o processamento de malhas), o *Geometry Assembler* para processar as geometrias, e o conversor e interpolador para processar os pixéis.

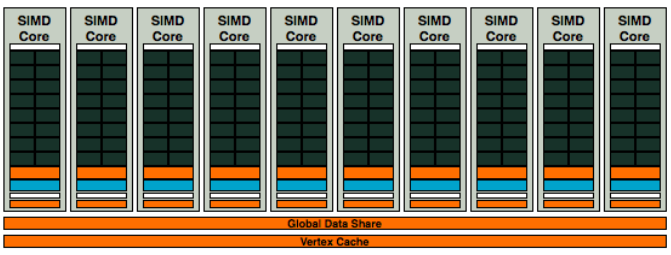


Figura 2.16: Organização dos *stream processors units* [71]

O *Thread Dispatch Processor* tem a responsabilidade de fazer a gestão das *threads* e manter todos os núcleos de processamento ocupados

6 O primeiro dispositivo proprietário da atual AMD foi o contador lógico AM2501 em 1970. A antiga empresa ATI que foi adquirida pela AMD Inc, produziu o seu primeiro controlador gráfico e respetiva placa em 1985 [68].

realizando as três operações em simultâneo.

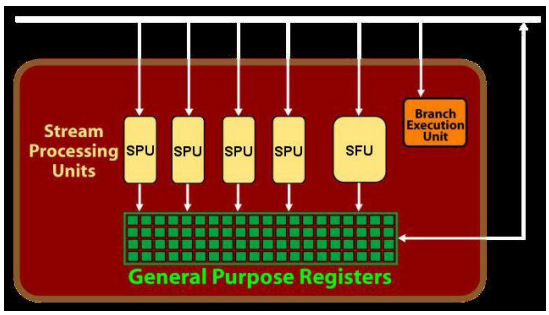


Figura 2.17: MSPU da arquitetura RV770 [70]

As *stream processor units* (SPU) estão organizadas em quatro matrizes, cada uma com 80 unidades e atuam segundo o padrão SIMD. A unidade fundamental de processamento, a SPU, é constituída por cinco unidades aritméticas e lógicas (ALUs) numa arquitetura superescalar, capazes de executar uma instrução de multiplicar-adicionar (MAD) por cada *clock* de relógio. A AMD equipou a R600 com 4 unidades de textura independentes que podem aceder às caches de vértices L1 (32KB) e L2 (256KB), podendo atingir uma taxa de transferência de 105,6 GB/s. Dispõe ainda de 4 unidades de renderização, capazes de produzir 4 pixéis por *clock* para o *frame buffer* e ainda realizar testes de profundidade e suavização.

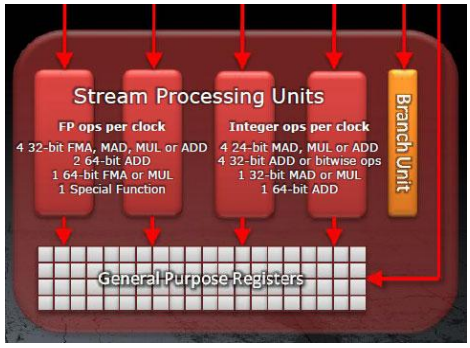


Figura 2.18: MSPU da arquitetura Cayman [72]

Seguiu-se a arquitetura RV770, com um novo arranjo interno: 10 SIMD *cores*, cada um com 16 *clusters* de 5 MSPUs (*Multi Stream Processing Units*), num total de 800 SPUs. Cada MSPU (Figura 2.17) tem quatro SPU para processamento de operações simples e um SPU com funcionalidades acrescidas, todos partilhando um mesmo grupo de registos.

Cada SIMD *core* tem acesso a 16KB de memória local para partilha de dados entre os *threads*, uma unidade própria de controle, 4 unidades de textura dedicada, acesso à *cache* L1 e comunica com os outros SIMD *cores* por 16KB de memória partilhada. As caches foram redesenhadas para que, a *cache* referente aos vértices fica separada, as caches L1 guardam apenas dados para cada SIMD *core* e as caches L2 ficam alinhadas com os canais de memória do *host* [41].

A arquitetura RV870, também designada por *Cypress*, foi a sucessora da RV770; essencialmente, esta arquitetura duplicou os recursos ou seja, passou dos 800 SPUs para os 1600 SPUs, eliminou as *fixed-functions* relativas aos interpoladores e acrescentou essa funcionalidade aos processadores dos *shaders*, garantindo ainda maior precisão. A organização dos SIMD *cores* e dos seus componentes internos mantiveram-se, alterando-se apenas a dimensão das memórias locais e globais, que duplicaram. Tem suporte para operações atómicas de 32 *bits* e possui mecanismos de sincronização por *hardware* (semáforos) e o seu pico de processamento é de 2,7 TFLOPS em precisão simples e 544 GFLOPS em precisão dupla [42]. Seguiu-se a arquitetura *Cayman* [43], muito semelhante à *Cypress*, exceto no consumo de energia, que foi muito melhorado, e no desempenho, que aumentou mais uma vez.

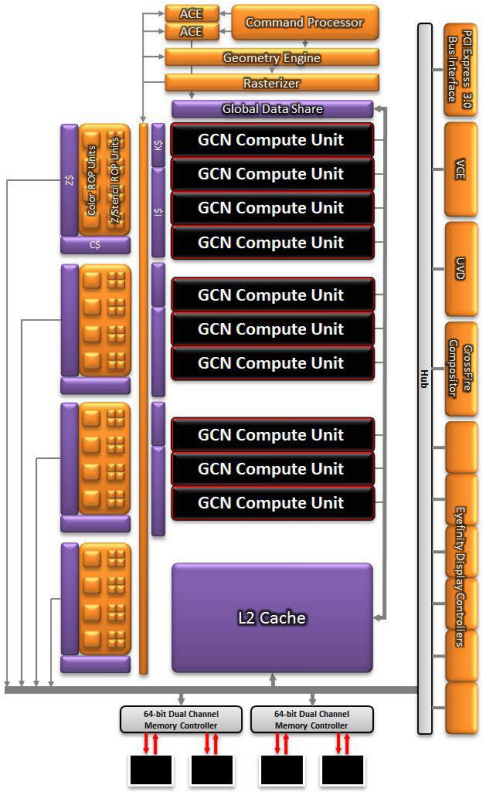


Figura 2.19: Arquitetura GCN [73]

Em 2011, a AMD concebeu uma nova arquitetura, *Graphics Cores Next* (GCN), para alcançar desempenhos de 1 e 4 TFLOPS, respetivamente, em precisão dupla e simples. Possui um processador de comandos responsável por receber os comandos referentes API de alto nível e mapeá-los para processamento nos diferentes *pipelines*, e ainda para realizar a coordenação do *pipeline* de renderização. As duas *Asynchronous Compute Engines* (ACE) que operam individualmente, são responsáveis pela gestão do processamento, pelo *scheduling* e alocação de recursos. Estas podem-se sincronizar e comunicar utilizando *cache*, memória ou 64KB da memória global partilhada (*Global Data Share* da Figura 2.19).

Define uma nova unidade de processamento que é designada por *Compute Unit* (*GCN Compute Unit* na Figura 2.20) que implementa um novo conjunto de instruções mais simples e que oferece melhor desempenho que os anteriores SIMD *cores*. No limite pode possuir até 32 unidades de processamento (2048 *stream processors*). As novas unidades de processamento são compostas por quatro SIMD independentes para processamento vetorial, no qual cada uma destas é composta por 16 SP que executam de forma independente uma só operação para cada fluxo.

Os SIMD possuem uma combinação de recursos privados e partilhados, sendo que os registos, o *buffer* das instruções e a ALU vetorial são recursos privados garantindo maior desempenho de utilização. Outros recursos como o *front-end*, *branch-unit* e a *cache* dos dados são partilhados de forma a garantir melhor eficiência energética.

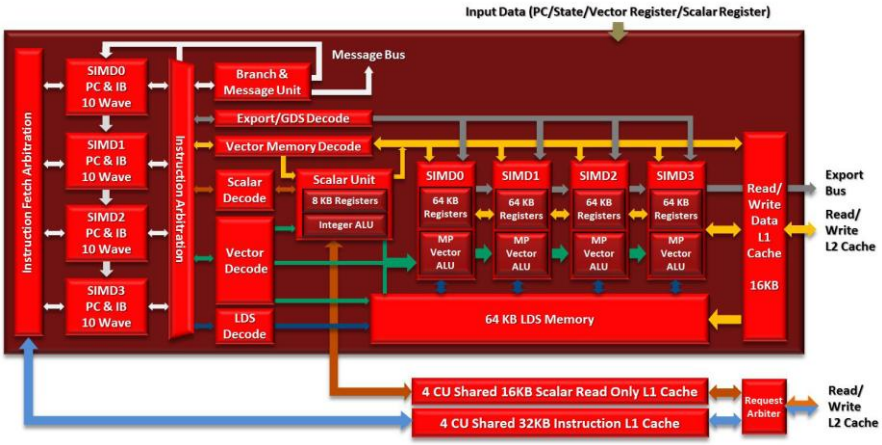


Figura 2.20: Arquitetura de uma CU[39]

A arquitetura GCN garante a coerência da *cache* e introduz o conceito de memória virtual através de combinação de *hardware* e suporte do driver, tornando-a compatível com arquitetura x86. Isto significa que pode trocar dados entre a CPU e a GPU e abre caminho para um espaço de endereçamento único, partilhado por CPUs e GPUs, levando a que se partilhe dados em vez de os copiar, vital para o desempenho e a eficiência energética. Incorpora ainda uma *I/O Memory Management Memory (IOMMU)* que mapeia de forma transparente endereços X86 para a GPU ou seja, podem aceder facilmente à memória paginável da CPU sem a sobrecarga de tradução de endereços para mover os dados.

Ao nível das comunicações possui interface PCI Express™ 3.0 usufruindo de taxas de transferência de 32GB/s, descodificador dedicado UVD3 que fornece suporte para formatos MPEG-4 e *DivX*, *Video Codec Engine (VCE)* com implementação por *hardware* da codificação H.264 capaz de garantir vídeo a 1080p a 60 *frames/s*.

Síntese das arquiteturas NVIDIA e ATI

Apesar da NVIDIA ter lançado o conceito de GPU em primeiro lugar, a AMD/ATI acompanhou de perto e desenvolveu uma tecnologia equivalente para obter os mesmos desempenhos. Tendo como base de partida para o desenvolvimento das GPUs o *pipeline* gráfico, é natural que as suas arquiteturas sejam semelhantes, relativamente aos componentes que as compõe, aos recursos utilizados, à tecnologia de produção utilizada e até os desempenhos obtidos.

Não obstante das suas semelhanças, os melhores avaliadores são os que efetivamente as utilizam quer no âmbito mais doméstico e empresarial, relativamente aos *desktops*, como também nos supercomputadores destinados ao processamento de alto desempenho. No uso doméstico e mais orientado para a indústria dos jogos, as avaliações das GPUs incorporadas no vasto leque de placas gráficas, são feitas por especialistas ligados a web*sites* especializados que utilizando *benchmarks* bem conhecidos e que confirmam que as duas indústrias progridem lado a lado apesar da ligeira vantagem obtida por uma ou outra [44] [45] [46]. No âmbito científico, nota-se uma maior utilização de dispositivos NVIDIA relativamente aos da AMD, no processamento de algoritmos e cálculos mas também aqui, a ambas as partes, são atribuídos os méritos de melhor desempenho [47] [48] ou equivalente [49]. No âmbito da HPC e observando a lista dos Supercomputadores TOP500 (Tabela 2.2), nota-se que os dispositivos da NVIDIA estão em maior número e possuem melhores desempenhos.

Tabela 2.2: Listagem de supercomputadores que utilizam aceleradores [21]

Accelerator/CP Family	Count	System Share (%)	Rmax (GFlops)	Rpeak (GFlops)	Cores
N/A	446	89.2	148,187,171	200,527,111	13,894,107
Nvidia Fermi	31	6.2	12,106,171	24,481,254	905,202
Intel Xeon Phi	11	2.2	42,113,382	67,790,175	3,830,503
Nvidia Kepler	8	1.6	20,094,200	30,957,401	645,340
ATI Radeon	3	0.6	938,700	1,832,963	62,832
Hybrid	1	0.2	196,234	262,560	8,412

2.2.2 Modelos de programação

Com a generalização e disseminação da utilização das GPUs e outros dispositivos de processamento, pretendeu-se de igual forma desenvolver modelos de execução que escalassem. Até à presente data, as abordagens utilizadas foram: utilização de plataformas como CUDA e *OpenCL*, programação baseada em diretivas como o *OpenMP* e *OpenACC*, algumas bibliotecas como FFT e BLAS e modelos baseados em MPI. Pretende-se abordar nas próximas secções as que mais diretamente se relacionam com o objeto deste trabalho: CUDA, *OpenCL* e *OpenACC*.

***Compute Unified Device Architecture* (CUDA)**

CUDA é uma plataforma *hardware/software*, proposto pela NVIDIA em 2006 e que permite executar programas desenvolvidos em C, C++, Fortran (e outras linguagens) sobre GPUs NVIDIA. Vocacionada para o processamento paralelo, pretende resolver problemas complexos mais rapidamente e de forma eficiente, dividindo-os em problemas de menor dimensão que podem ser resolvidos paralelamente e de forma independente, por blocos. Trata-se de um modelo de programação escalável que faz proveito da paralelização dos GPUs *manycore*, no qual o desafio consiste em construir aplicações que usem esse paralelismo de forma transparente fazendo recurso a uma grande quantidade de *threads* [33, p. 4] [50, p. 2].

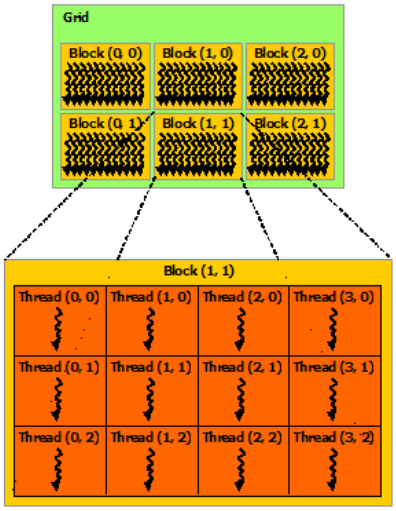


Figura 2.21: Organização, *threads*, blocos e grids [33].

Um programa CUDA pode combinar código sequencial executado no CPU (o *host*) e código paralelo a ser executado na GPU (o *device*), invocando funções paralelas chamadas *kernels*, que executam *threads*. *Kernels* são então funções que, quando chamadas, são executadas paralelamente e instanciadas por diferentes *threads* CUDA. O compilador ou o programa, organiza as *threads* em blocos, e estes em *grids* (Figura 2.21).

Cada *thread* executa uma instância do *kernel*, possui um *program counter*, registos, memória privada, dispositivo de input e output. A memória privada é utilizada para os transbordos de registos, chamadas de funções e variáveis automáticas do tipo *array*. As *threads* acedem para operações de leitura e escrita à memória global, à memória partilhada e registos, e apenas em operações de leitura à memória constante e de texturas. Os CUDA *threads* são mais simples que os CPU *threads*. A cada *thread* em execução é atribuído um *thread* ID, guardado na variável *built-in threadIdx*, ID esse válido enquanto durar a execução do *kernel*. A *threadIdx* é uma variável vetorial que pode ser definida a 1, 2 ou 3 dimensões.

Um bloco (de *threads*) é um conjunto de *threads* que executam concorrentemente e que podem cooperar entre si usando memória partilhada e usar mecanismos de sincronização tais como barreiras; cada bloco possui um identificador único, dentro de uma *Grid*, identificador esse que é também uma variável vetorial, *blockIdx*, e que pode ser definida a 1, 2 ou 3 dimensões. Existe um número máximo de *threads* por bloco, sendo que atualmente os blocos podem conter mais que 1024 *threads* [33, p. 8]. A dimensão do bloco de *threads* pode ser identificado por uma variável de sistema *blockDim*. O número de blocos de *threads* de uma *Grid* é normalmente determinado pelo tamanho de dados a processar, ou pelo número de *cores* do GPU.

Uma *Grid* é um *array* de blocos que executa um determinado *kernel*, lê e escreve dados na memória global e sincroniza chamadas de *kernels* dependentes. As *grids* partilham os resultados no espaço de memória global após sincronização global do *kernel*. Um *kernel* pode ser executado em múltiplos blocos de *threads* logo o número total de *threads* em execução corresponde ao produto do número de *threads* por bloco pelo número de blocos.

Quando se desenha uma aplicação CUDA, há que considerar o aspeto da gestão da memória, já que existe a memória do *host* e a do *device*, e o acesso é, em termos de latência, não-uniforme. O *host* pode transferir dados de, e para o *device*: para a memória global, a memória constante e a memória de texturas. O *runtime* CUDA fornece funções para reservar (pode ser reservada de forma linear ou matricial) e libertar espaço de memória no *device*, tais como *cudaMalloc(), cudaFree(),* e *cudaMalloc3d(),* bem como funções para transferir dados entre *host* e *device*, tais como *cudaMemcpy(), cudaMemcpy2D(), cudaMemcpy3D()*.

***Open Computing Language* (OPENCL)**

É um padrão aberto para programação paralela de propósito geral, em GPUs, CPUs e outros dispositivos de processamento, oferecendo portabilidade e eficiência. O *OpenCL* consiste numa API para coordenar o processamento entre processadores heterógenos, utilizando a linguagem C (subconjunto da ISO C99) e C++ (*OpenCL C++ wrapper API*), num ambiente de processamento bem especificado. O desenvolvimento do *OpenCL* foi iniciado pela Apple mas a sua gestão cabe ao *Khronos Group* [51].

Trata-se de uma multiplataforma que lida com diversas tecnologias atualmente existentes, pelo qual as suas funcionalidades já são suportadas por grandes empresas, entre as quais a NVIDIA [52], AMD [53] e Intel [54]. Sendo uma das suas características a sua portabilidade, os programas que fazem recurso ao *OpenCL* podem atingir desempenhos diferenciados, conforme a tecnologia utilizada, obrigando ao desenvolvimento de um vasto leque de possíveis configurações e testes, conforme as capacidades disponíveis em cada recurso.

O seu modelo de execução é semelhante ao CUDA e é constituído por um programa que é executado no *host* e *kernels* que são executados nos *devices* do *OpenCL*. Quando um *kernel*

é chamado, este é instanciado por *work items* que correspondem aos *CUDA cores.* Existe um espaço de índices global que identifica cada *work item* e que mapeia os dados que vão ser operados por cada um. Os *work items* formam *work groups* que correspondem no CUDA aos blocos de *threads*. Os *work items* são identificados num intervalo de índices de dimensão global (*NDRange*), comparável à *Grid* no CUDA. Operações de sincronização entre *work items* só ocorrem dentro de um *work group* através da utilização de barreiras de sincronização ou no final da execução do *kernel*. Cada *device* é constituído por um ou mais *comput units* (correspondem ao SM do CUDA ou aos *cores* da CPU ou qualquer unidade de coprocessamento) e cada *comput unit* é composto por um ou mais *processing elements* (PEs) que correspondem aos SP no CUDA, sendo a unidade básica de processamento.

À semelhança das arquiteturas anteriores, existe uma hierarquia de memórias sendo elas: global, constante. local e privada. O *host* pode reservar espaço de memória dinamicamente nas memórias global, constante e local e pode realizar operações de leitura e escrita na memória global e constante. A memória global, de maior dimensão e maior latência relativamente às restantes, é acessível para leitura e escrita a todos os *work items* dos vários *work groups.* A memória constante, reservada estaticamente apenas pode ser lida pelos *work items.* A memória local (equiparada à memória partilhada no CUDA) pode ser lida e escrita por *work items* dentro de um *work group.* A memória privada (equiparada á memória local e registos do CUDA), apenas pode ser reservada estaticamente e pode ser lida e escrita por um único *work item.*

***OpenACC Application Programming Interface* (API) [55]**

A *OpenACC* API é um *standard* que define um conjunto de diretivas de compilação, bibliotecas e variáveis de ambiente que podem ser utilizadas para escrever programas paralelos em C, C++ e FORTRAN que podem depois ser executados em arquiteturas heterogéneas que incluem CPUs e aceleradores (GPUs ou outros). A especificação *OpenACC* foi inicialmente desenvolvida pela Portland Group (PGI), Cray Inc., e NVIDIA, com o apoio da empresa CAPS.

Uma das principais diferenças relativamente aos modelos anteriormente apresentados é que o *OpenACC* baseia-se, tal como o *OpenMP*, em diretivas de compilação. O código, em tudo semelhante a um código sequencial, dispõe de uns marcadores especiais que indicam ao compilador que uma determinada sequência deve ser despachada para um acelerador e processada em paralelo.

O modelo de execução baseia-se num *host* e aceleradores que suportam três níveis de paralelismo; numa aproximação *top-dow* os aceleradores possuem conjuntos de unidades de execução, cada uma capaz de executar múltiplas *threads*, sendo que cada *thread* é capaz de executar operações sobre vetores. Assim, quando é lançado um programa onde existe uma região de código que foi anotado, um *kernel* é executado, síncrona ou assincronamente, do

lado do *device*. A execução do *kernel* segue um modelo *fork-join* em que são criados *gangs* de *threads* (blocos de *threads* no CUDA) que por sua vez são agrupados em *worker groups* (*warps* no CUDA) que operam sobre vetores.

As memórias do *host* e dos *devices* são tratadas separadamente, e assume-se que o *host* não tem acesso direto à memória do *device*. As transferências de dados não são implementadas de forma explícita como nos modelos anteriores, sendo essa preocupação passada para o compilador que reserva, copia e liberta a memória necessária. A consistência da memória entre vários *workers* é fraca em virtude de não existirem mecanismos de sincronização entre estes e não se garantir a coerência de dados em memória.

O *OpenACC* não assume qualquer capacidade de sincronização exceto para os *fork-joins* das suas *threads*. Assim, quando é lançado algum trabalho ele é executado sem interrupções até terminar.

O *OpenACC*, comparativamente aos modelos anteriores, traz alguns benefícios tais como: a implementação do programa pode ser escrita inicialmente de forma sequencial e posteriormente ser anotada para processamento paralelo; detalhes de transferência de dados, *caching*, lançamento de *kernels*, *thread scheduling*, mapeamento do paralelismo, etc., ficam a cargo do compilador e do *runtime*; para programas já desenvolvidos não é necessário reescrever o código novamente mas apenas anotá-lo; o código quando compilado por compiladores não *OpenACC*, é compilado e executado como código sequencial; para efeitos de *debug* é muito mais fácil compreender o que se está a passar quando se visualiza código sequencial. Contudo, o paralelismo no *OpenACC* é baseado no compilador ao contrário das anteriores onde o paralelismo é explicitado na própria implementação.

2.2.3 Ambientes de desenvolvimento

Com o CUDA, veio um ambiente de desenvolvimento que permite aos programadores trabalharem com linguagens de alto-nível (C, C++, Fortran, Python), bibliotecas e APIs (Figura 2.22) tendo como objetivo garantir uma curva de aprendizagem baixa.

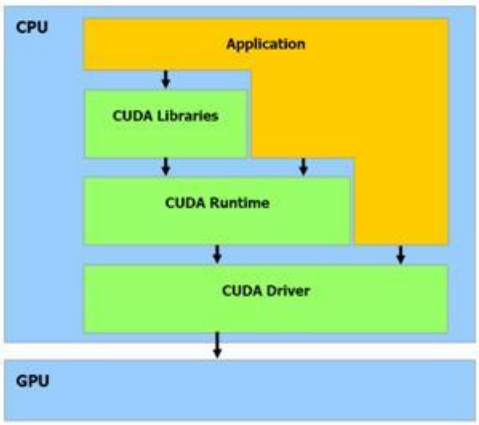


Figura 2.22: Camadas CUDA

Um *kernel* pode ser programado usando diretamente o *assembly* da arquitetura, chamado PTX; contudo o usual é utilizar uma linguagem de maior nível de abstração, por exemplo, C. O CUDA fornece um conjunto mínimo de extensões à linguagem C, de tal forma que um *kernel* é visto como uma função, com uma sintaxe própria para declarar e definir blocos e *grids*. O código é então compilado com o `nvcc`.

A biblioteca do *runtime* é construída sobre a API do driver CUDA, este também acessível às aplicações permitindo o acesso a conceitos de baixo nível como por exemplo, contexto e os módulos [33, p. 14].

O `nvcc` é um driver de compilação que converte ficheiros `.cu` em `.c` para o *host* e em CUDA *assembly* ou instruções binárias para os dispositivos. Suporta um conjunto de parâmetros para a otimização, tais como número de registos por *kernel*, normalização de números em vírgula flutuante e simplificação das respetivas operações, entre outras, que permitem executar mais rapidamente à custa da precisão e correção.

A compilação pode decorrer em modo *offline* ou em tempo de execução. No modo *offline*, os ficheiros fonte podem ter tanto código do *host* como do *device* e, aquando a compilação, ocorre a separação do código a ser executado. O código do dispositivo é compilado e transformado em PTX ou num objeto binário (*cubin object*). Relativamente ao *host*, o compilador modifica as chamadas aos *kernels* substituindo a sintaxe CUDA (extensões à linguagem C) por chamadas a funções do *runtime* CUDA. O código modificado é deixado na linguagem C e será compilado por uma outra ferramenta, e.g., `gcc`, num outro estágio de compilação. No caso da compilação ser em tempo de execução, qualquer código PTX carregado por uma aplicação é compilado para código binário pelo *device* driver. Este processo aumenta o tempo de carregamento mas permite que as aplicações beneficiem das melhorias dos compiladores que surgem com novos dispositivos. Quando o driver compila algum código PTX, este automaticamente faz uma cópia para uma *cache* do código binário resultante de forma a evitar repetições posteriores de compilação do mesmo código que não tenha sofrido alterações. Esta *cache*, que pode ter dimensões dos 32MB aos 4GB, possui mecanismos que invalidam as cópias logo que haja atualizações produzidas [33, p. 15].

O *CUDA runtime* está implementado numa biblioteca dinâmica, a biblioteca *cudart*. Assume que o sistema é composto por um *host* e um dispositivo e que cada um possui a sua própria memória. Não existe uma função de inicialização explícita, mas ela resulta da chamada de uma função do dispositivo. Durante a inicialização o *runtime* do CUDA cria um contexto para cada dispositivo que é partilhado por todos as *threads* da aplicação *host*. Este contexto é válido enquanto o *runtime* estiver a executar e não é visível para a aplicação. A chamada `cudaDeviceReset()` destrói o contexto primário do *device* mas o *host* pode continuar a executar e, caso chame uma nova função do *device*, é criado um novo contexto.

Cada dispositivo possui capacidades de processamento específicas (Tabela 2.3) que correspondem à disponibilidade de funções por hardware e operações disponíveis.

Tabela 2.3: *Compute Capability* por arquiteturas

Compute Capability	1,0	1,1	1,2	1,3	2,0	2,1	3,0	3,5	5,0
Threads / Warp	32	32	32	32	32	32	32	32	32
Warps / SM	24	24	32	32	48	48	64	64	64
Threads / SM	768	768	1024	1024	1536	1536	2048	2048	2048
Thread Blocks / SM	8	8	8	8	8	8	16	16	32
Shared Memory / SM (Kb)	16	16	16	16	48	48	48	48	64
Max Shared Memory / Block (Kb)	16	16	16	16	48	48	48	48	48
32-bit registers / SM	8192	8192	16384	16384	32768	32768	65536	65536	65536
Register Allocation Unit Size	256	256	512	512	64	64	256	256	256
Register Allocation Granularity	block	block	block	block	*warp*	*warp*	*warp*	*warp*	*warp*
Max Registers / Thread	124	124	124	124	63	63	63	255	255
Shared Memory Allocation Unit Size	512	512	512	512	128	128	256	256	256
Warp Allocation Granularity	2	2	2	2	2	2	4	4	4
Max Thread Block Size	512	512	512	512	1024	1024	1024	1024	1024
Shared Mem Configurations (Kb)	16	16	16	16	16/48	16/48	16/32/48	16/32/48	16/32/48
Warp register allocation granularities					64/128	64/128	256	256	256

2.2.4 Curvas de aprendizagem e produtividade

Os modelos de programação referidos anteriormente embora pretendam oferecer, de forma transversal, uma forma simples e fácil de implementar o paralelismo de execução, apresentam características diferenciadas que têm impacto quer na aprendizagem quer na produtividade.

A plataforma CUDA e o padrão *OpenCL*, lidam com o paralelismo obrigando a uma implementação explícita do paralelismo o que exige bons conhecimentos das arquiteturas internas, modelos de execução e modelos de memória. Apesar de apresentarem algumas semelhanças, na sua composição, organização e funcionamento, o padrão *OpenCL* pode ser considerado mais complexo em virtude de utilizar recursos de processamento heterogéneos e ter que lidar com as capacidades de processamento de cada recurso. Por outro lado, a sua portabilidade torna-se uma vantagem quando consegue produzir resultados em diferentes dispositivos ao invés do CUDA que apenas funciona com as GPUs NVIDIA.

O *OpenACC*, diferencia-se do CUDA e do *OpenCL*, por permitir reutilizar uma implementação sequencial, requerendo "apenas" a tarefa de anotar o código nas partes onde pode ser paralelizado, com duas vantagens imediatas: a primeira, é que deixa à responsabilidade do compilador e do *runtime* tarefas associadas à paralelização, libertando o programador de ter de conhecer os detalhes da tecnologia utilizada; a segunda, todo o trabalho de implementação sequencial que já tenha sido efetuado não tem que ser novamente reescrito,

aumentando desta forma a sua produtividade.

2.2.5 Expectativas de desempenho

No âmbito do processamento sequencial e extensível ao processamento paralelo, o senso comum diria que, com mais processadores obtêm-se maior desempenho. Contudo, tal pode não se verificar pois além do número de processadores está dependente da possibilidade das tarefas serem paralelizadas. Para uma boa abordagem dessa problemática, em primeiro lugar é necessário compreender como é que as aplicações podem escalar, depois pode-se definir expectativas de desempenho, de seguida definem-se planos e estratégias, quer no plano do *hardware*, como na implementação da paralelização do problema a resolver. Aspetos como latência e larguras de banda da memória, quantidade de dados transferidas entre memórias, paralelismo de instruções, ramificações de execução e sincronização de *threads*, influenciam igualmente o desempenho das aplicações.

No âmbito da programação paralela, um programa é constituído por uma parte que é executada sequencialmente (*s*) e por outras partes que são executadas em paralelo (*p*). Tendo como base a Lei de Amdhal [56] [50, pp. 5-7] , assumindo que s+ $p = 1$, a razão entre o tempo de execução para um processador e o tempo de execução de n processadores dá-nos o *Speedup*, ou seja:

$$Speedup = \frac{s+p}{s+\frac{p}{n}} = \frac{1}{s+\frac{1-s}{n}} = \frac{n}{ns+(1-s)}$$

onde *n* quantifica o número de processadores/*cores* a executar[7]. Assim, podemos deduzir as seguintes conclusões:

- s = 0 e n = #p, processadores/*cores*, então *Speedup = #p*, indicando que todo o código seria paralelizável e o *Speedup* seria diretamente proporcional ao número de processadores/*cores* disponíveis;
- s = 1 e n = #p, processadores/*cores*, então *Speedup = 1*, indicando que a execução será sequencial independentemente do número de processadores/*cores*.
- Considerando que as atuais GPUs possuem entre 1500 e 2000 *cores* , defina-se n = 1500 e os seguintes valores de s:
    - *s = 0,5* então *Speedup = 1,998*
    - *s = 0,1* então *Speedup = 9,940*
    - *s = 0,05*: então *Speedup = 19,74*

7 Apesar das atuais CPUs serem *multicores* e possuírem várias *threads*, comparativamente com o número de cores das atuais GPUs acaba por ser um número desprezável.

Pode-se concluir que o *Speedup* é muito sensível à variável *s.*

A Lei de Gustafson-Barsis [57] [50, pp. 5-7] defende que *p* e *n* dependem um do outro e que portanto, para usar mais processadores é necessário aumentar o trabalho do problema executado em paralelo. Assim define:

$$Speedup = n + (1 - n) \times s$$

onde *n* quantifica o número de processadores/*cores* a executar. Assim, pode-se ainda deduzir as seguintes conclusões:

- s = 0 e n = #p, processadores/*cores*, então *Speedup = #p*, o *Speedup* seria diretamente proporcional ao número de processadores/*cores* disponíveis;
- s = 1 e n = #p, processadores/*cores*, então *Speedup = 1*, indicando que a execução será sequencial independentemente do número de processadores/*cores*.
- Considerando que as atuais GPUs possuem entre 1500 e 2000 *cores* , defina-se n = 1500 e os seguintes valores de s:
    - *s = 0,5* então *Speedup = 5,5*
    - *s = 0,1* então *Speedup = 9,1*
    - *s = 0,05*: então *Speedup = 9,55*

  Apesar de ser mais otimista, continua-se a verificar que o *Speedup* é muito sensível à variável *s.*

Em resumo, pode-se dizer que, para a atual dimensão do número de *cores*, só se atinge grandes acréscimos de desempenhos quando se consegue modular um problema e se consegue paralelizar o seu processamento.

A tecnologia já se encontra num bom estádio de desenvolvimento e os problemas podem ser modelados de tal forma que, podem ser mapeados em recursos computacionais que ofereçam processamento paralelo. Importa agora verificar se os atuais modelos conseguem responder a estes requisitos de desempenho. Até à data, alguns testes realizados demonstram que CUDA e *OpenCL* possuem desempenhos comparáveis e que estão na mesma ordem de grandeza apesar de *OpenCL* oferecer maior portabilidade [58] [59]. O *OpenACC*, encontra-se numa fase inicial de desenvolvimento e ainda não existem muitos testes comparativos com as restantes implementações. Alguns estudos realizados [60] [61] apresentam um potencial de crescimento no que se refere ao seu desempenho mas possui desde já vantagens na facilidade de compreensão e implementação.

2.3 Otimização de computações em GP-GPUs

2.3.1 Conhecer detalhadamente a arquitetura de execução

A arquitetura GPU da *NVIDIA* está construída tendo por base uma matriz de *cores* SM(X) com capacidade de executar múltiplas *threads*. Quando um programa invoca um *kernel*, este constitui uma *Grid* de blocos de *threads* distribuídos pelos SM(X). Os blocos de *threads* são executados concorrentemente em cada SM(X) e as *threads* dentro de cada bloco são também executados concorrentemente. Os CUDA *cores* e outras unidades de execução dos SM(X) executam uma *thread* de instruções. Quando um bloco de *threads* termina a execução das instruções, dá lugar a outro bloco de *threads* para ser executado [33, pp. 63-64].

Os SM(X), para gerirem a vasta quantidade de *threads* distribuídos pelos diferentes blocos, seguem uma arquitetura do tipo *Single-Instruction Multiple-Thread (SIMT)*. Os SM(X) criam, gerem, e despacham a execução de instruções, agrupando os *threads* em grupos de 32 que se designam por *warps*. As *threads* de um *warp* possuem o seu próprio contexto (*instruction address pointer* e registos), executam de forma paralela e as suas execuções podem seguir ramos distintos. Os *warps* ainda se podem dividir em *half-warps* e *quarter-warps*. Quando é atribuído um bloco de *threads* a um SM, este sequencia as *threads* pelo seu ID de forma crescente e divide em grupos de 32, constituindo assim os *warps* que por sua vez serão escalonados para execução por um *warp* scheduler.

Os *warps* executam uma instrução comum de cada vez e obtém máxima eficiência quando não existem ramificações de execução nem dependência de dados. Caso ocorra divergências, a execução dos *threads* concordantes é serializada e apenas quando todos as ramificações forem executadas e convergirem voltam a executar em bloco. O contexto de execução de cada *warp* (*program counters*, registos, etc.) é mantido *on-chip* durante o seu tempo de vida e a mudança de contexto não tem grandes custos. O *warp* scheduler seleciona o *warp* que estiver disponível para ser executado.

Cada SM possui um conjunto de registos de 32 *bits* e memória partilhada que são divididos pelos *warps*. O número de blocos e *warps* em memória e em execução, depende do número de registos e memória partilhada disponível para cada multiprocessador. Existe também um limite máximo de blocos e *warps* em cada SM(X) que é fixado conforme a *compute capability* do *device*. Caso não haja recursos disponíveis dentro do SM(X), pelo menos para executar um bloco de *threads*, a execução do *kernel* falha.

2.3.2 Otimização em arquiteturas CUDA

Implementações corretas de paralelismo resultam num maior desempenho e rendimento das aplicações e dos recursos utilizados. Se no âmbito de programas sequenciais as melhorias de desempenho podem ser obtidas, de forma genérica, através de otimizações oferecidas pelos compiladores, utilização eficiente da hierarquia de memórias (caracterizada essencialmente por efeitos NUMA e de *caching*) e aperfeiçoamentos ao nível dos sistemas de operação e

dispositivos de I/O, já nos programas de execução paralela a otimização do desempenho é obtida pelo facto de se maximizar a execução paralela para obter a máxima utilização de recursos, otimizar a utilização das hierarquias de memória (estas caracterizadas não só pela presença de múltiplas *caches* mas também, ao contrário do que acontece com as arquiteturas SMP/NUMA, por espaços de endereçamento funcionalmente separados: memórias partilhada, constante, texturas), e otimizar a utilização das instruções de forma a ter o máximo rendimento destas [33, p. 66].

Maximizar a utilização

Por forma a maximizar a utilização, a aplicação deve ser estruturada de tal forma que, exponha o máximo paralelismo possível e que de forma eficiente faça o mapeamento desse paralelismo nos diversos componentes do sistema, mantendo-os ocupados o máximo tempo possível [33, p. 66].

A nível aplicacional deve-se atribuir as tarefas sequenciais ou de baixo nível de paralelismo ao *host* e as paralelizáveis aos *devices*. Para manter produtivamente ocupados, simultaneamente, o *host* e os *devices* pode-se utilizar funções assíncronas para, por exemplo, realizar transferência de dados entre memórias e executar um *kernel*. Também é natural que durante a execução de uma aplicação, os dados em memória sejam partilhados por *threads*, quer do mesmo bloco, quer de blocos diferentes. Quando a partilha é entre *threads* de um mesmo bloco, deve-se utilizar a memória partilhada desse bloco e invocar nesse mesmo *kernel* a função `_syncthreads()`; se a partilha é entre *threads* de blocos diferentes, deve-se então utilizar-se a memória global e usar *kernel* distintos, para escrita e para leitura, contudo deve-se redesenhar a solução para que a comunicação *inter-threads* ocorra dentro do mesmo bloco.

Ao nível dos *devices*, a aplicação deve maximizar a execução paralela entre os SM(X) e, tendo em atenção as *compute capability* dos *devices*, devem ser lançados o maior número possível de *kernels* concorrentes por *device* por forma a maximizar a sua utilização.

Ao nível dos multiprocessadores SM(X)s, a aplicação deve maximizar a execução paralela entre as várias unidades funcionais do multiprocessador, sendo que estas dependem do nível de paralelismo das instruções, que por sua vez está diretamente relacionado com o número de *warps* residentes e com a latência de execução de instruções. A latência resulta do número de ciclos necessários para que um *warp* esteja pronto para executar uma instrução, sendo que o ideal seja que a cada ciclo de relógio exista um *warp* a executar uma instrução. Obviamente, o referido "número de ciclos necessários para que um *warp* esteja pronto para executar uma instrução" está em si mesmo dependente da latência de acesso aos dados em memória, dos mecanismos de sincronização e ainda da *compute capability* do *device* relativamente ao número de ciclos necessários para carregar as instruções e número de ciclos

necessários para executar as instruções [33, pp. 67-68].

Relativamente ao número de blocos e *warps* residentes em cada SM(X)s, este depende da configuração de execução das chamadas *kernel*, dos recursos de memória partilhada disponíveis em cada SM(X) e dos recursos necessários ao *kernel*. Este número pode ser determinado com uma ferramenta aplicacional, "*CUDA Occupancy Calculator*", na qual calcula a taxa de ocupação, determinada pelo rácio entre o número de *warps* residentes e o número máximo de *warps* possíveis. Relativamente ao total de memória partilhada exigido por um bloco de *threads* é igual ao somatório da quantidade da memória reservada estática e dinamicamente[8]. Também o número de registos utilizados por um *kernel* pode ter um impacto significativo no número de *warps* residentes pois existe uma dependência tal que o número de *warps* residentes está dependente do número de registos necessários para a execução do *kernel*, da quantidade de *threads* (agrupados em *warps*) e do limite máximo de número de registos que o SM suporta. O número de registos utilizados ainda está dependente da utilização ou não, de operações de cálculo de dupla precisão. Para efeitos de otimização, o número de *threads* por bloco deve ser um múltiplo do tamanho dos *warps* de forma a não desperdiçar recursos com *warps* que não estão a ser utilizados [33, p. 69].

Os efeitos de desempenho da configuração de execução do *kernel* dependem do código do *kernel*, algo que só é possível aferir através da experimentação. A parametrização dos valores pode ter como base o tamanho dos registos, do tamanho da memória partilhada, a *compute capability* do *device*, o número de multiprocessadores e a largura de banda do *device* [33, p. 69].

Maximizar o rendimento das transferências de memória

De forma geral as transferências de dados, por necessárias que sejam, prejudicam o desempenho de uma aplicação uma vez que, geralmente, a computação não pode progredir enquanto a transferência se desenrola; uma exceção é quando se programa usando transferências assíncronas, que se conseguem sobrepor (*overlap*) com tarefas computacionais. Dito isto, as boas estratégias a aplicar são, por ordem: a) minimizar o número de transferências; b) minimizar o tempo de transferência; c) se possível, sobrepor computação com transferência de dados (Figura 2.23).

8 No caso dos devices com compute capability 1.x acresce a memória utilizada para a passagem dos argumentos das funções.

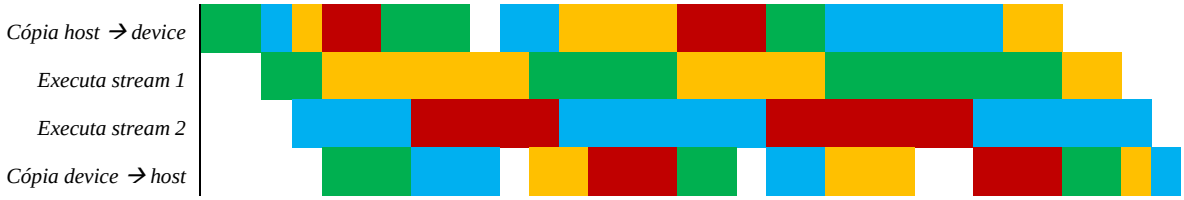


Figura 2.23: Concorrência de cópia de dados e processamento[9]

A tarefa (a) de minimizar o número de transferências decorre do desenho da aplicação, das estruturas de dados, e do mapeamento destas na(s) arquitetura(s) alvo – pode-se, por exemplo, criar estruturas de dados intermédias, manuseadas e destruídas no *device*, sem ter que as mapear no *host* ou transferir para este.

A minimização do tempo de transferência (b) pode conseguir-se combinando duas vertentes: usando caminhos com (1) maior largura de banda (LB), e de (2) menor latência. Por exemplo, no caso de aceleradores inseridos num *bus* de I/O, como é o caso de uma placa NVIDIA conectada ao PCI-e, a transferência de dados entre o dispositivo e a RAM do *host* é muito penalizadora pois incorre não só numa elevada penalização ao nível da latência de acesso ao *bus*, como ainda de uma LB diminuta quando comparada com a disponível no *bus* CPU/memória ou nos *buses* internos da placa aceleradora. Numa frase: devem evitar-se transferências entre o *host* e o *device*. Mas, como é impossível eliminá-las completamente, devem ser, se possível, otimizadas: i.e., o tempo total deve ser minimizado.

Para minimizar o tempo total de transferência ($T_{tot} = T_L + T_T$) devem realizar-se transferências de quantidades de dados que sejam suficientes para que o tempo de transferência propriamente dito (T_T) seja bastante superior às latências de preparação e finalização do processo de transferência (T_L); isso significa que, se a quantidade de dados a transferir for baixa, deve-se procurar agregar esses dados a outros, até conseguir um volume adequado (já que T_T = Volume/LB).

Minimizar a latência não é fácil, mas para sistemas que possuam um *front-side bus*, pode-se recorrer ao mecanismo de *page-locked host memory*, e às funções `cudaHostAlloc()` e `cudaFreeHost()`, para efetuar cópias concorrentes entre as memórias do *host* e do *device* e até, em alguns *devices*, mapear memória do *host* no espaço de endereçamento do *device*: neste caso, trata-se de uma forma de memória partilhada, e deixa de ser necessário reservar memória no *device* e fazer cópias entre *host* e *device* [33, p. 71]

A hierarquia de memória num dispositivo CUDA inclui duas classes de "memória", *on-*

[9] Cada cor representa as operações associadas a um conjunto de dados e.g., a representação inicial a verde indica que para um conjunto de dados foi realizada a cópia de dados do *host* para o *device* depois é executado um determinado procedimento sobre esses dados e por fim decorre a cópia dos dados do *device* para o *host*.

chip, que como o nome indica, inclui todos os tipos de memória que existem no interior da pastilha (*chip*) NVIDIA e *off-chip*, que inclui todos os outros. A classe *off-chip* pode ainda ser subdividida em duas: *on-board*, que inclui as memórias que não estando dentro do *chip* estão todavia na placa aceleradora, e *off-board*, que se resumem à memória do *host* (RAM). Num dispositivo NVIDIA CUDA encontramos então os seguintes tipos de memória (ordenados por ordem decrescente da LB e crescente da latência):

- *on-chip*: registos, *caches* de nível 1 (L1), memória partilhada;
- *off-chip, on-board*: *cache* de nível 2 (L2) e memória global;
- *off-board*: memória do *host* (RAM).

Como as *caches* não são visíveis em termos do modelo de programação, a nossa atenção no que se refere às larguras de banda concentra-se inevitavelmente nos registos (LB da ordem da dezena de TB/s), memória partilhada (poucos TB/s), memória global (poucas centenas de MB/s) e memória do *host* (LBs da ordem de alguns GB/s, mas latências 100x superiores às da memória global) [62, pp. 109-131].

Contudo, para conseguir as larguras de banda anteriormente referidas (pico da LB), é preciso que os padrões de acesso sejam ótimos, o que exige um enorme esforço no desenho das estruturas de dados e nos algoritmos; senão, vejamos, a título de exemplo: a memória partilhada está dividida em módulos de igual dimensão designados bancos, permitindo que estes possam ser acedidos simultaneamente. Quando ocorrem conflitos de acesso (simultâneo) a um mesmo banco, estes são serializados via *hardware*, e o desempenho diminui [33, p. 74]

Maximizar o rendimento das instruções

A maximização do desempenho pode ser também obtida acelerando a execução das instruções propriamente ditas; para o conseguir podem utilizar-se várias estratégias distintas, das quais destacamos: a) usar instruções que exijam menos ciclos de relógio para serem executadas; b) reduzir os tempos de espera.

São exemplos de (a) os casos de utilização da precisão simples em vez de dupla nos cálculos em vírgula flutuante, da utilização de instruções nativas, suportadas por *hardware*, em vez de se usarem "funções de biblioteca", ou da utilização de funções intrínsecas na multiplicação, divisão, raiz quadrada e operações trigonométricas em vez de genéricas (sendo que as últimas garantem resultados rigorosamente idênticos quando executadas no *device* ou no *host*).

São exemplos de (b) os casos de utilização de instruções de controlo do fluxo como *if, switch, do, for, while,* que têm um impacto direto no rendimento das instruções pelo facto de poderem provocar a divergência na execução de *threads* dentro de um mesmo *warp*. Se tal acontecer, obriga à serialização das instruções e à utilização de mecanismos de sincronização, o que implica maior número de instruções e possibilidade de colocar *threads* no estado *idle*. Partindo do princípio que as divergências terão que acontecer e conhecendo a forma

determinística como os *warps* são distribuídos, estas podem ser minimizados.

Em resumo, toda a lógica de implementação da aplicação deve ter em vista a execução paralela e concorrente nos diferentes recursos disponíveis, a minimização da latência nos acessos aos dados privilegiando os caminhos que tiverem maior largura de banda, oferecerem acessos coalescentes, e no qual as instruções exigidas para o processamento sejam de rápida execução sem descurar a sua precisão e correção.

2.3.3 Usar bibliotecas de alto desempenho

A NVIDIA oferece mais de uma dezena de bibliotecas para auxiliar e acelerar o desenvolvimento de aplicações, aumentando a produtividade [63]. Importa aqui salientar além das principais, as que eventualmente poderão ser utilizadas neste trabalho.

A ***biblioteca runtime***, é uma biblioteca dinâmica, que não requer inicialização, aplicável tanto ao *host* como aos *devices*, que oferece funções para gestão dos *devices*, *streams*, memória e tratamento de erros. Existem duas APIs, a *cuda_runtime_api.h* de suporte à linguagem C e a *cuda_runtime.h*, para o C++.

CUDA Basic Linear Algebra Subprograms (CUBLAS) é uma implementação da biblioteca *Basic Linear Algebra Subprograms* (BLAS) que recorre ao *NVIDIA runtime* para permitir ao programador controlar o acesso aos recursos de processamento das GPUs. A sua utilização é simples, bastando para tal que a aplicação reserve memória para matrizes e/ou vetores no espaço da memória da GPU, preencha com dados, chame as funções desejadas da API e envie os resultados para o *host*. Possui compatibilidade com a linguagem Fortran, permite a execução de chamadas de funções da biblioteca de forma assíncrona, pode ser empregue em diferentes fluxos e faz retorno de informação de erros para efeitos de depuração. Não é *thread-safe*, sendo que a sua utilização deve ser desencadeada apenas por uma *thread* do lado do *host*.

CUDA Math é uma coleção de funções matemáticas padrão de alta precisão, desenhadas para um bom desempenho em GPUs.

CUDA Sparse (cuSPARSE) é uma biblioteca que fornece um conjunto de rotinas básicas de álgebra linear para matrizes esparsas

2.3.4 Ferramentas de análise e desempenho

A NVIDIA oferece 3 ferramentas de análise de desempenho [64]: *Visual Profiler*, *Command Line Profiler* e *nvprof*.

O ***Visual Profiler*** é uma ferramenta gráfica que permite visualizar e otimizar o desempenho das aplicações CUDA. Disponibiliza um conjunto de informações de análise, deteta estrangulamentos no decorrer da execução, identificando os potenciais problemas e aponta possíveis soluções. Organizado por vistas fornece informações sobre atividades

executadas pela CPU e pela GPU, as suas durações, linhas temporais referentes a processos, *threads*, chamadas às API em tempo de execução, chamadas à API do driver, transferência de dados entre os diferentes tipos de memória, informações sobre o *kernel*, *streams*, etc. A análise pode ser efetuada a todo o programa ou apenas a parte deste.

A ferramenta ***nvprof*** permite recolher e observar dados de *profilers* a partir da linha de comandos. Permite igualmente a recolha de linhas temporais referentes às atividades desenvolvidas pela CPU e GPU incluindo execução de *kernel*, transferência de dados e chamadas às APIs. Permite a recolha de valores de contadores por *hardware*. Como limitação tem o facto de não ter capacidade de recolha de métricas e apenas consegue fazer o *profile* das aplicações diretamente lançadas. Permite três modos de operação: sumário, *trace* da GPU e API e, sumário/*trace* de Eventos. O modo sumário é o modo por omissão e produz um output simples, onde resulta uma linha por cada função *kernel* (tempo total de todas as instâncias, bem como o tempo mínimo, médio e máximo) e os tipos de memória utilizado na cópia realizada pela aplicação. O modo *trace* da GPU e/ou API, pode ser executado separadamente ou em conjunto. Quando é realizado o *trace* da GPU, descreve por linhas de tempo as atividades desenvolvidas pela GPU por ordem cronológica. É fornecida informação por cada *kernel* e instância de cópia de memória, bem como informação detalhada dos parâmetros do *kernel* e desempenho da memória partilhada utilizada. No modo API-*trace* mostra a linha de tempo de todas as chamadas CUDA feitas em tempo de execução e ao driver do API. No modo de eventos corresponde a contadores de valores que são recolhidos durante a execução do *kernel*. Dentro deste, no modo *trace*, os eventos são mostrados por cada execução do *kernel*, de forma agregadas por todas as unidades da GPU. O output gerado pode ser visto no *Visual Profiler*.

O ***Command Line Profiler*** é uma ferramenta de medição de desempenho e pesquisa de potenciais oportunidades de otimização. Permite obter via de linha de comandos informações sobre tempo de execução do *kernel* e operações de transferência de dados entre memórias. As opções do *profiling* são controladas por meio das variáveis de ambiente e de um ficheiro de configuração. O formato de dados de saída é em texto, chave-valor ou formato CSV. O output gerado pode ser visto no *Visual Profiler*.

ParaView – A Parallel Visualization Application é uma ferramenta *open-source*, multiplataforma, para visualização e análise de conjuntos de dados a duas e três dimensões associadas à componente tempo. Possui uma interface gráfica que permite visualizar e manipular dados, permitindo no caso do presente trabalho, verificar o comportamento do objeto em análise. O processamento de dados e os componentes de renderização estão assentes numa arquitetura paralela de memória distribuída, modular e escalável em que muitos processadores operam sincronamente em diferentes partes dos dados. Tal arquitetura permite trabalhar eficientemente com grandes volumes de dados e efetuar uma visualização gráfica dos resultados produzidos [65].

3

3 Trabalho Realizado

3.1 Descrição do problema a otimizar

Conforme referido inicialmente, o problema a otimizar envolve a interação de enormes conjuntos de partículas, (sejam interações destas entre si ou com as superfícies limite) o que exige uma série de cálculos, mais ou menos elementares, sobre um grande volume de dados, pois trata-se do cálculo sucessivo das posições das partículas no plano tridimensional, com a inclusão da componente tempo, o que permite verificar a variação do seu comportamento dinâmico. Para efeitos de cálculo consideram-se as partículas como esferas de raios semelhantes, sendo que as irregularidades na sua geometria são modeladas com a introdução de fatores de forma. Os potenciais contactos das partículas são determinados no início de cada passo da simulação por um algoritmo que é executado na CPU, tendo em conta o centro das partículas e o seu movimento de rotação e assumindo-se que não existem interpenetrações.

A simulação é executada para um determinado número de passos especificado por um parâmetro da aplicação; em cada passo, os cálculos ocorrem por iterações sucessivas e terminam quando o erro admissível para a simulação (especificado por um outro parâmetro) é atingido ou, se não se verificar, quando o número limite de iterações é atingido (outro parâmetro da aplicação). Nos cálculos têm-se em consideração a posição inicial da partícula, a sua velocidade e aceleração, a massa e a força. A formulação do problema baseia-se no deslocamento das partículas, limitado que está pelos contactos potenciais a que estão sujeitas; define-se portanto uma função-objetivo, sujeita a restrições. Para a resolução do problema é utilizado um algoritmo iterativo, que opera sobre um grande volume de dados, determinam-se mínimos locais e globais, atualizam-se as posições (dados) e realizam-se testes de convergência.

A implementação existente foi elaborada na linguagem C com extensão CUDA da NVIDIA e testada sobre tecnologia da arquitetura Fermi, mais propriamente em placas Tesla C2050 e M2075. A execução decorre tanto na CPU como na GPU. Como exemplo de tarefas

executadas pelo CPU destacamos as entradas de parâmetros e leitura de dados de ficheiros, reservar memória no *host* (i.e., na RAM acessível ao CPU) e/ou no *device* (i.e., na RAM acessível ao GPU), copiar os dados para a memória do *device* e depois em cada passo, detetar os possíveis contactos entre partículas, utilizando para tal um algoritmo de triangulação (Triangulação de *Delaunay*). Obtidos esses possíveis contactos, estes são copiados para a GPU e voltam apenas a ser recalculados no início do próximo passo. Do lado da GPU, são executados, de forma sequencial, um conjunto de *kernels* que representam as operações matemáticas necessárias para a resolução do problema, como mostra, na página seguinte, a Figura 3.1.

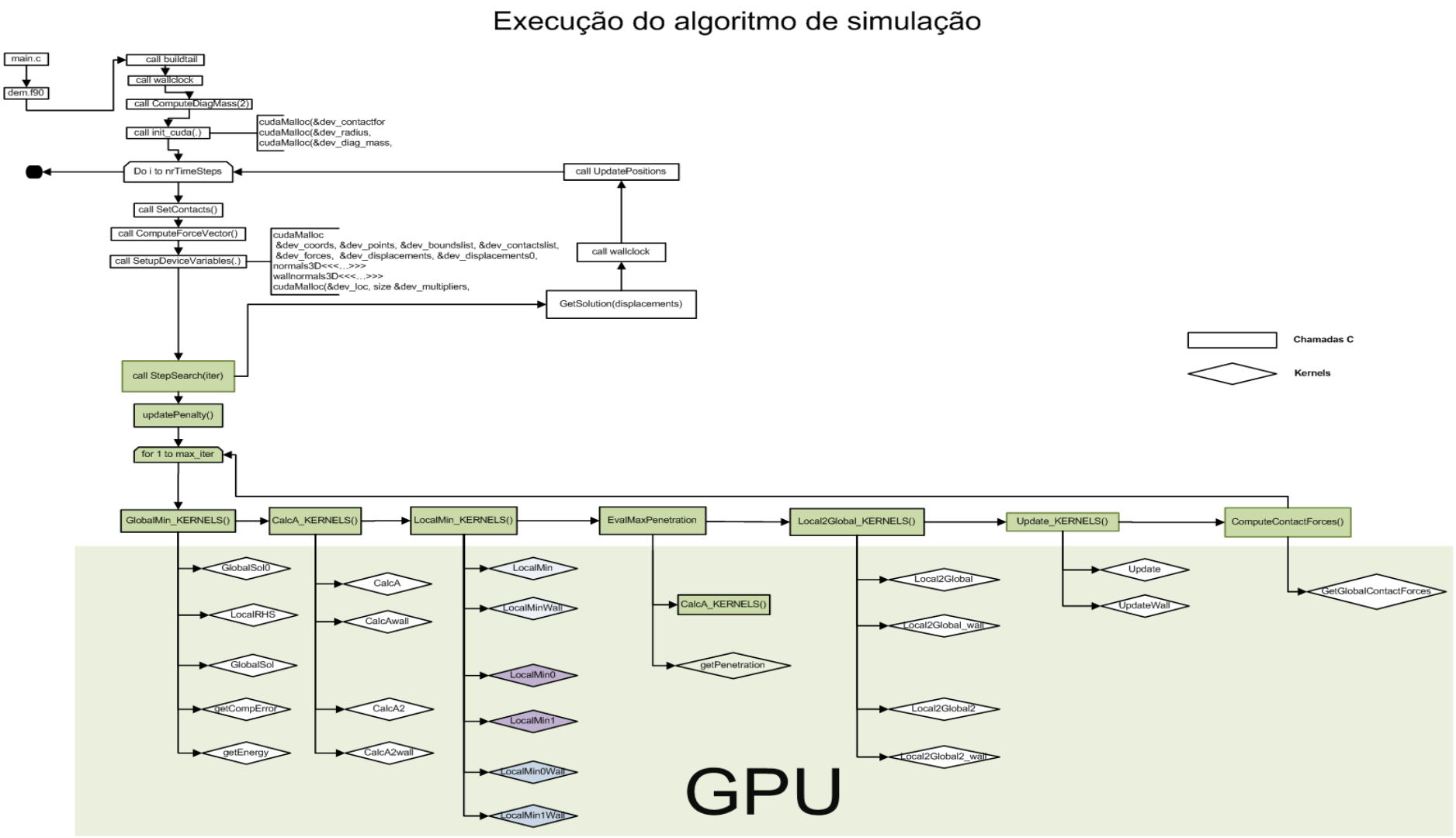


Figura 3.1: Fluxograma de execução do algoritmo de simulação.

3.2 Avaliação inicial da aplicação

A avaliação inicial da aplicação foi realizada em três etapas: inicialmente avaliou-se a evolução do tempo de execução e consumo de memória variando a amostra de dados; em seguida realizou-se o profiling da aplicação de forma global e numa ultima etapa o *profiling* a partes específicas, a alguns *kernel*s, da aplicação.

3.2.1 Avaliação do tempo de execução e consumo de memória

Houve necessidade de avaliar o impacto no tempo de execução da aplicação quando se variam dois parâmetros nomeadamente: a quantidade de partículas utilizada e o número de passos realizados. Conforme se pode observar na Figura 3.2 e na Figura 3.3, o tempo de execução cresce linearmente quer para uma variação incremental de 100 mil partículas quer com uma variação incremental de 5 *TimeSteps*.

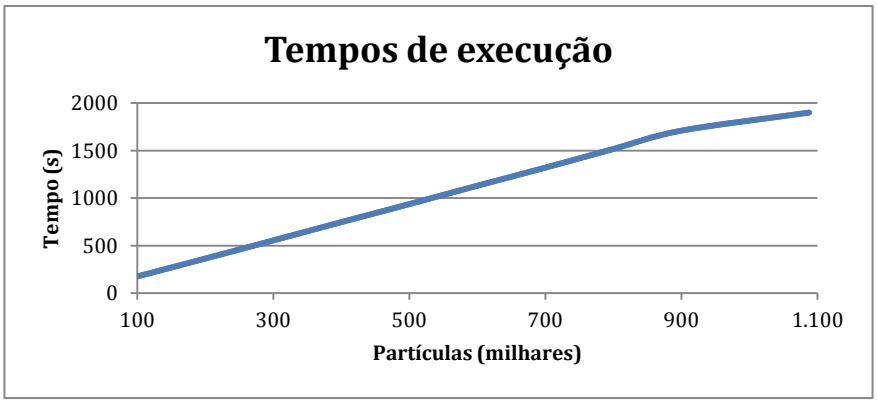


Figura 3.2: Evolução do tempo de execução variando o número de partículas processadas

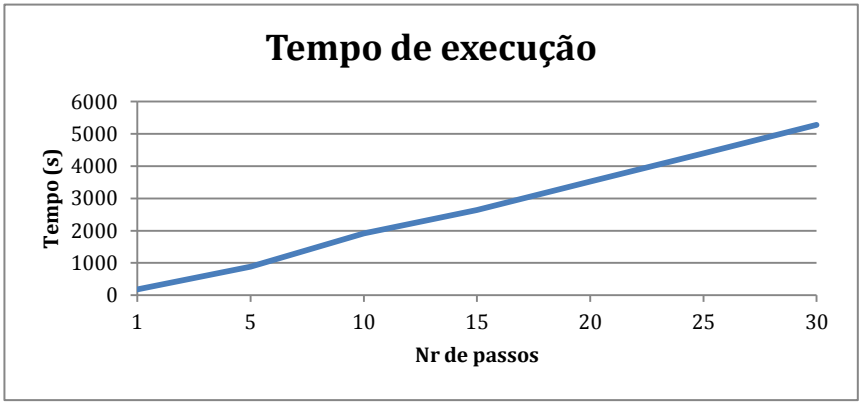


Figura 3.3: Evolução do tempo de execução variando o número de *TimeSteps*

Também foi importante avaliar o consumo de memória utilizado pela aplicação tendo em conta o número de partículas utilizadas e constatou-se que o consumo de memória cresce linearmente com o número de partículas utilizadas (Figura 3.4).

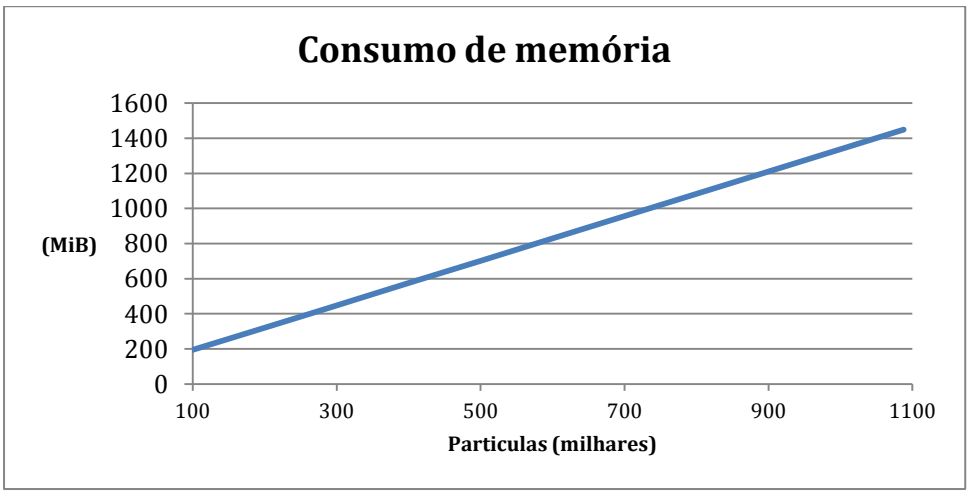


Figura 3.4: Evolução do consumo de memória variando o número de partículas

3.2.2 Profiling da aplicação

O *profiling* inicial da aplicação foi realizado com recurso à ferramenta *NVIDIA Visual Profiler* que possui um vasto conjunto de métricas e que disponibiliza informação detalhada dos resultados obtidos, sugerindo possíveis otimizações. O *profiling* foi realizado para uma amostra de 1 milhão de partículas, para um só *timestep* com 10 iterações apenas (para minimizar o tempo de recolha, processamento de dados, e apresentação dos "relatórios") e a aplicação foi parametrizada para incluir todas as variáveis possíveis de simular (atrito partícula-partícula, partícula-superfície limite e resistência ao rolamento) por forma a possibilitar a avaliação de todos os *kernels* que a aplicação contém. Assim, obtivemos informações relativamente à transferência dos dados entre memórias, operações concorrentes, utilização dos processadores e desempenho global dos *kernels*.

Transferência de dados e concorrência

No que se refere à concorrência de operações, nomeadamente cópia de dados entre memórias (global, partilhada, *caches* L1 e L2) em simultâneo, existência de operações de cópia de dados em simultâneo com operações de processamento (do *host* e do *device*), bem como *throughputs* conseguidos, pode-se observar pela Figura 3.5 que as percentagens de eficiência são nulas ou reduzidas.

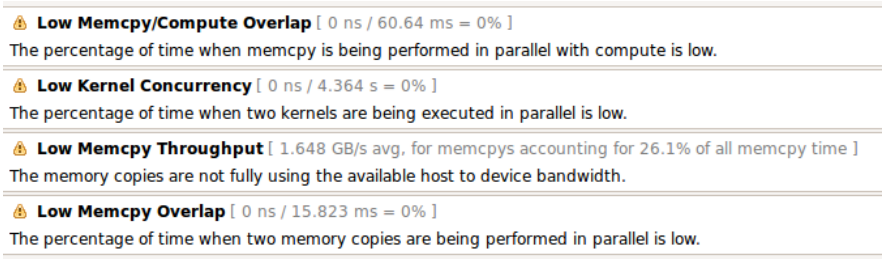


Figura 3.5: Profiling da aplicação sobre cópia de dados e concorrência

Tais resultados devem-se essencialmente à não paralelização das operações e ao facto de serem utilizadas operações síncronas. Possíveis soluções de otimização passam pela utilização de operações assíncronas recorrendo a *streams* que permitem intercalar e sobrepor operações de cópias de dados e processamento de *kernels*. Tal é possível quando a GPU utilizada disponha de *copy engines*, como é o caso da C2050 utilizada, permitindo por exemplo em simultâneo, cópias de dados do *host* para o *device*, do *device* para o *host* e execução de *kernels*. De igual forma, através da utilização de vários *streams* é possível executar concorrentemente vários *kernels*, o que poderá ser interessante se houver multiprocessadores com baixas taxas de utilização.

Observa-se também que a largura de banda do barramento não é totalmente utilizada, o que poderá ter várias causas, sendo uma destas a taxa de *page-faults* que ocorre quando há acessos a memória. Se tal se vier a verificar, poderemos remediar esta situação fazendo o *pinning* da memória.

Utilização dos processadores

Relativamente à utilização da GPU e seus processadores, que se pode observar na Figura 3.6, note-se que os 6% referidos como baixa utilização da GPU decorrem dos parâmetros usados para obter o profiling – 1 timestep e 10 iterações – o que acaba por distorcer o peso relativo da parte sequencial (CPU, onde se gastam aproximadamente 68 segundos) vs. paralela (GPU).

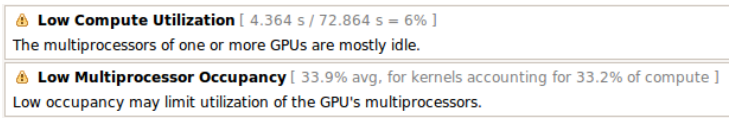


Figura 3.6: Profiling da aplicação relativo à utilização dos multiprocessadores

Existe também uma baixa taxa de ocupação[10] dos multiprocessadores (33.9%). A taxa de ocupação depende de 3 fatores, designadamente: do número de *threads* definidos por bloco, da quantidade de memória partilhada utilizada em cada bloco e do número de registos utilizados por *thread*. Eventualmente, poder-se-á aumentar a taxa de ocupação fazendo variar estes fatores.

Relativamente à percentagem de tempo de utilização dos processadores da GPU, poderemos observar na Figura 3.7 que os *kernels* que consomem mais tempo são respetivamente: *Update, GlobalSol, CalcA, Local2Global* e *LocalMin*. Serão estes os *kernels* objeto de avaliação inicial e de otimizações futuras.

[10] A taxa de ocupação é a razão entre o número de *warps* ativos e o número total de *warps* possíveis que o multiprocessador pode suportar.

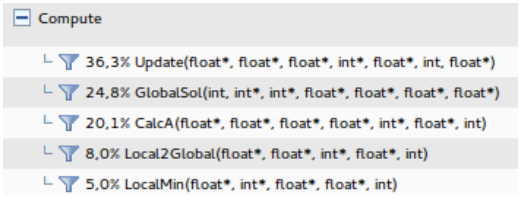


Figura 3.7: Percentagem de utilização dos multiprocessadores da GPU

Desempenho dos *kernels*

Relativamente ao desempenho global dos *kernels*, a ferramenta de *profiling* da NVIDIA apresenta a eficiência de utilização das transferências de dados em operações de *load* e de *store* para a memória global e memória partilhada[11] e da eficiência da execução dos *warps*.

Conforme se pode observar na parte superior da Figura 3.8, verifica-se que a eficiência obtida para operações de *load* e *store* para memória global é baixa, melhorando para as operações em memória partilhada. Os resultados para a baixa eficiência no acesso à memória global devem-se a dois fatores: ao não alinhamento dos dados na execução dos *warps* e a padrões de acesso não uniformes. Interessa que as operações de acesso à memória global, quer de leitura ou escrita, sejam o mais coalescentes possível para que os acessos a memória pelas *threads* de um *warp* tenham o mínimo de transações de blocos. Deve-se igualmente procurar utilizar ao máximo a memória partilhada, pois obter-se-ão menores latências. Dependendo das capacidades de processamento da GPU, poderão ainda ser utilizadas *caches* nas memórias L1 e/ou L2 nas quais as *threads* dos *warps* beneficiarão do alinhamento de dados para as suas operações. Ainda relativamente à eficiência das operações na memória partilhada, os resultados obtidos devem-se à concorrência de acessos das operações de leitura e escrita sobre os mesmos bancos de memória, pois quando assim acontece as operações são serializadas.

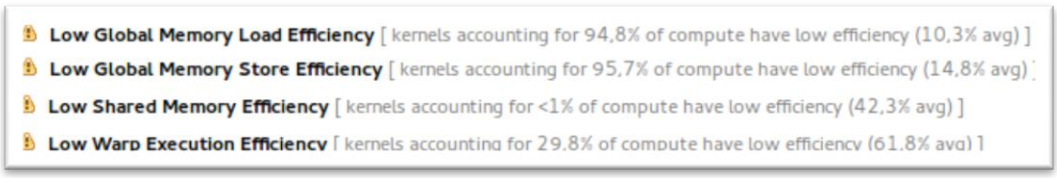


Figura 3.8: Profiling da aplicação relativamente à eficiência das operações sobre a memória

Relativamente à eficiência de execução dos *warps*[12] (parte inferior da Figura 3.8) conseguem-se obter resultados satisfatórios mas estes dependem da divergência da execução, resultado da utilização de instruções de controlo de fluxos tais como: *if*, *switch*, *do*, *for* e *while* que afetam o *throughput* de instruções executadas pelas *threads* dentro de um mesmo *warp*. Quando assim acontece e porque as *threads* partilham o mesmo *program counter*, os fluxos

[11] A eficiência das operações é definida pela razão entre o número de *bytes* requeridos e o número de *bytes* efetivamente transacionados nas operações de *load* e *store*.

[12] Média das *threads* ativas em cada *warp* executado.

de instruções a executar são serializados. Será desejável minimizar este tipo de instruções, obtendo maior eficiência de execução dos *warps* e consequentemente aumentando a taxa de ocupação dos multiprocessadores.

3.2.3 Caracterização dos *kernels*

Observado o *profiling* da aplicação no geral, interessa agora analisar cada *kernel* individualmente e verificar o seu desempenho relativamente ao processamento, latência, utilização de memória e execução. Dada a extensão desta análise remete-se para anexo (Anexo A) a consulta mais detalhada sobre o assunto, descrevendo-se de seguida os aspetos que queremos evidenciar.

Tabela 3.1: Quadro resumo do *profiling* dos *kernels*

	Update	*GlobalSol*	*CalcA*	*Local2Global*	*LocalMin*
Grid Size	[42,1,1]	[41,1,1]	[14,1,1]	[14,1,1]	[28,1,1]
Block size	[512,1,1]	[448,1,1]	[512,1,1]	[512,1,1]	[384,1,1]
Register/Threads	26	23	63	63	44
Shared Memory/Block (KiB)	2	1,574	42	42	22,5
Global load Efficiency	6,5%	4,3%	6,9%	9,6%	9,4%
Global Store Efficiency	12,5%	16,7%	12,5%	12,5%	12,5%
Shared Efficiency	92,3%	99,7%	100,0%	100,0%	100,0%
Warp Execution Efficiency	100,0%	73,3%	93,5%	100,0%	44,1%
Occupancy Archieved	65,7%	86,9%	33,2%	33,1%	24,6%

A taxa de ocupação é um bom indicador da utilização dos recurso de processamento do *device*, podendo esta ser limitada pelo número de registos utilizados, pela memória partilhada reservada e pelo número de *threads* em cada bloco. Apesar de nem sempre se garantir melhores desempenhos quando se atinge boas taxas de ocupação, até porque o desempenho pode depender de outros fatores, como por exemplo da latência de carregamento de dados/instruções e/ou execução de instruções, é um bom princípio configurar os *kernels* para maximizar a utilização dos recursos que a GPU possui e assim as possíveis soluções de otimização passam por reestruturar os *kernels* de modo a reduzir o número de registos e quantidade de memória partilhada necessárias para a execução do *kernel*.

Como se pode observar pela Tabela 3.1 apenas o *kernel GlobalSol* possui uma boa taxa de ocupação, seguido do *Update* a 2/3, estando os restantes a 1/3 das suas capacidades.

Numa análise relativa ao número de registos por bloco, pode-se verificar que os *kernels CalcA* e *Local2Global* estão no limite do número máximo de registos por bloco, que é de 63, possivelmente acarretando *spill* dos dados para memória local; estão também próximos do limite máximo da memória partilhada por bloco (48KB) sendo, portanto, possíveis candidatos a otimizações relativamente a estes dois fatores; também os *kernels LocalMin* e *Update* poderão ser alvo de otimizações atendendo à sua baixa/média taxa de ocupação apesar de não terem atingido nenhum dos fatores que limitam a taxa de ocupação.

Evidencia-se a boa eficiência obtida resultante das operações sobre a memória partilhada nos *kernels* acima referidos ainda que, no *kernel Update*, essa eficiência seja ligeiramente menor.

De igual modo, observa-se a boa eficiência obtida da execução dos *warps* sendo que, aqueles que obtiveram menor eficiência estão relacionados com divergências que decorrem da implementação (por exemplo, no *kernel GlobalSol*, página 54, na linha 18, o *profiler* reportou uma divergência de 100%) .

De forma transversal a todos os *kernels* verificou-se uma baixa eficiência nas operações de *load* e *store* sobre a memória global, devendo-se tal situação ao facto dos acessos a esta memória pelos *warps* não serem coalescentes. Sabendo à partida que muitos dos acessos não são contíguos é eventualmente possível aumentar a eficiência "desligando" a cache L1 e usando *caching* ao nível da L2, que tem linhas de 32 bytes (contra os 128 bytes da L1).

Observando agora a Tabela 3.2, na qual a linha designada "*Utilization*" foi retirada dos dados de profiling, constata-se que a utilização, pela aplicação, das larguras de banda disponíveis nos diversos níveis da hierarquia de memória só é alta no caso do acesso à memória global (no manual da placa C2050 a largura de banda assinalada é de 144GB/s).

Tabela 3.2: Quadro resumo da utilização de largura de banda disponível no *device*

		Update	*GlobalSol*	*CalcA*	*Local2Global*	*LocalMin*
L1 /Shared Memory	*Transactions*	178672601	147674674	145209934	96131100	47609084
	Bandwidth (GB/s)	134,475	158	194,526	306,662	246,509
	Utilization	Baixa	Baixa	Baixa	Média/Baixa	Média/Baixa
L2 Cache	*Transactions*	617525652	519384284	513850238	189054602	864355550
	Bandwidth (GB/s)	124,381	138,813	187,27	171,809	124,722
	Utilization	Média/Alta	Média/Alta	Alta/Max	Alta	Média/Alta
Device memory	*Transactions*	534013421	475762649	300486013	111639545	60319120
	Bandwidth (GB/s)	107,56	127,155	109,511	101,455	87,037
	Utilization	Alta	Alta/Máx	Alta	Alta	Média/alta

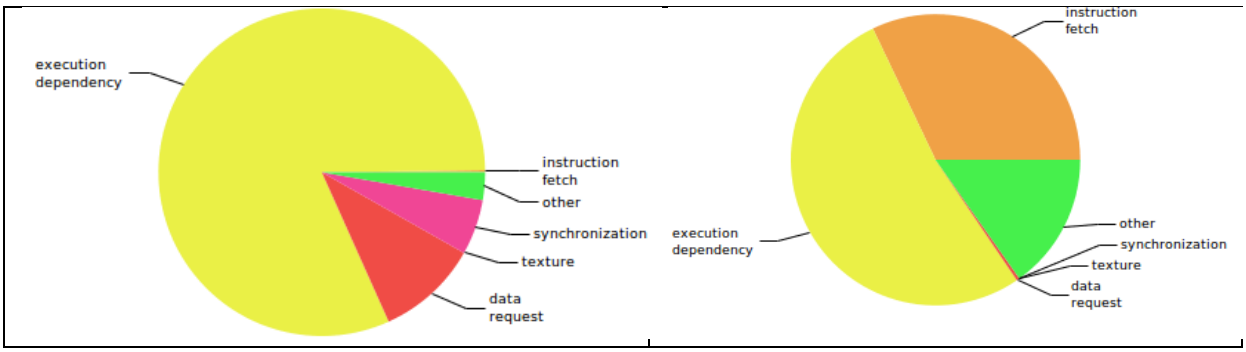


Figura 3.9: Origem da latência de execução das instruções do *kernel GlobalSol* (à esquerda) e do *kernel LocalMin* (à direita)

Constatou-se que todos os *kernels* apresentavam enormes latências na execução das instruções em consequência de dependências de dados (espera por resultados que dependem da execução de outras instruções). Verificou-se também que o *kernel* LocalMin apresentava alguma latência resultante das instruções a serem executadas ainda não estarem disponíveis.

Relativamente ao tipo de instruções executadas pelos *kernels* verifica-se que são essencialmente de duas naturezas: ou operações aritméticas simples, ou associadas a transações de acesso à hierarquia de memórias, e em particular a acessos às memórias L2 e global.

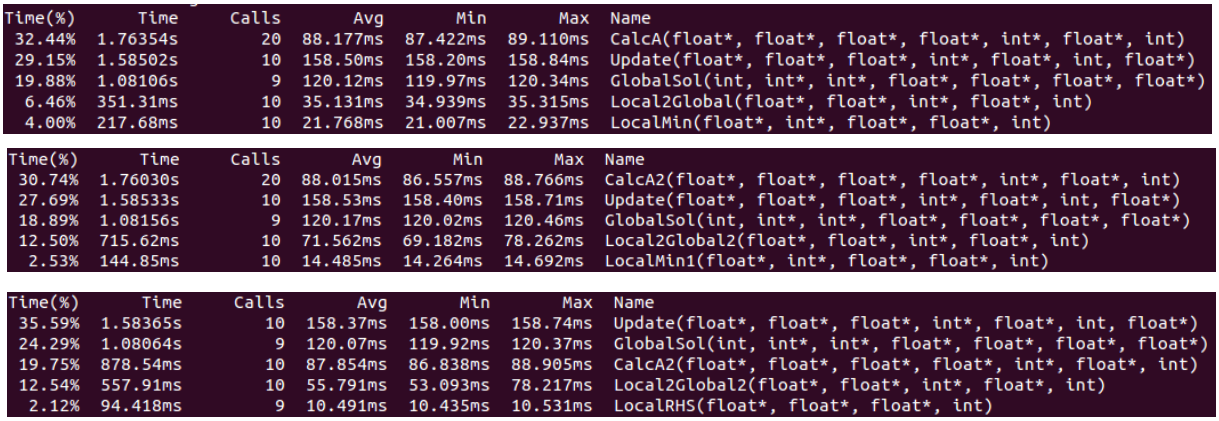
Kernels especializados

A implementação possui um conjunto de *kernels* que tratam todas as situações possíveis de interação, e dois conjuntos adicionais especializados, que são muito semelhantes aos anteriores, mas são ainda mais simples por excluírem alguns dos tipos de interação; estes, que designaremos doravante por *kernels* especializados, estão listados na Tabela 3.3.

Tabela 3.3: Família de *kernels* especializados

***Kernel* principal**	***Kernel* semelhante**
Update	*Update0*
GlobalSol	*GlobalSol0*
CalcA	*CalcAwall,* *CalcA2, CalcA2wall*
Local2Global	*Local2Global_wall,* *Local2Global2, Local2Global2_wall*
LocalMin	*LocalMinWall,* *LocalMin0, LocalMin0wall,* *LocalMin1, LocalMin1Wall*

Relativamente à execução desses *kernels* especializados, eles possuem características semelhantes e tempos de execução da mesma grandeza que os *kernels* genéricos. Tal facto pode ser confirmado através dos *profiles* exibidos na Figura 3.10 e, como tal, serão abordados seguindo a mesma metodologia que os *kernels* principais.



Time(%)	Time	Calls	Avg	Min	Max	Name
32.44%	1.76354s	20	88.177ms	87.422ms	89.110ms	CalcA(float*, float*, float*, float*, int*, float*, int)
29.15%	1.58502s	10	158.50ms	158.20ms	158.84ms	Update(float*, float*, float*, int*, float*, int, float*)
19.88%	1.08106s	9	120.12ms	119.97ms	120.34ms	GlobalSol(int, int*, int*, float*, float*, float*, float*)
6.46%	351.31ms	10	35.131ms	34.939ms	35.315ms	Local2Global(float*, float*, int*, float*, int)
4.00%	217.68ms	10	21.768ms	21.007ms	22.937ms	LocalMin(float*, int*, float*, float*, int)

Time(%)	Time	Calls	Avg	Min	Max	Name
30.74%	1.76030s	20	88.015ms	86.557ms	88.766ms	CalcA2(float*, float*, float*, float*, int*, float*, int)
27.69%	1.58533s	10	158.53ms	158.40ms	158.71ms	Update(float*, float*, float*, int*, float*, int, float*)
18.89%	1.08156s	9	120.17ms	120.02ms	120.46ms	GlobalSol(int, int*, int*, float*, float*, float*, float*)
12.50%	715.62ms	10	71.562ms	69.182ms	78.262ms	Local2Global2(float*, float*, int*, float*, int)
2.53%	144.85ms	10	14.485ms	14.264ms	14.692ms	LocalMin1(float*, int*, float*, float*, int)

Time(%)	Time	Calls	Avg	Min	Max	Name
35.59%	1.58365s	10	158.37ms	158.00ms	158.74ms	Update(float*, float*, float*, int*, float*, int, float*)
24.29%	1.08064s	9	120.07ms	119.92ms	120.37ms	GlobalSol(int, int*, int*, float*, float*, float*, float*)
19.75%	878.54ms	10	87.854ms	86.838ms	88.905ms	CalcA2(float*, float*, float*, float*, int*, float*, int)
12.54%	557.91ms	10	55.791ms	53.093ms	78.217ms	Local2Global2(float*, float*, int*, float*, int)
2.12%	94.418ms	9	10.491ms	10.435ms	10.531ms	LocalRHS(float*, float*, float*, int)

Figura 3.10: *Profiles* dos *kernels* especializados

3.3 Processo de otimização

A metodologia seguida para otimizar a implementação foi abordar cada um dos *kernels* individualmente, tendo como base de partida os resultados da avaliação inicial e o tempo médio de execução do *kernel*, e realizar as otimizações que se consideraram convenientes; esta primeira fase é relatada nas secções 3.3.1 a 3.3.6. Após a otimização individual de cada *kernel*, foram realizadas no conjunto da aplicação as otimizações globais expostas na secção 3.3.7.

3.3.1 *Kernel* Update

Os possíveis pontos a otimizar deste *kernel* são: melhorar a eficiência das operações sobre a memória global, aumentar a taxa de ocupação caso corresponda a melhor desempenho, reduzir a latência de execução de instruções relacionada com a dependência de dados. As configurações iniciais e o tempo de execução inicial são os constantes na Tabela 3.4.

Tabela 3.4: Configuração e valores de partida do *kernel* Update

#Block	#Threads/B	Reg	ShMem (B)	Occupancy (%)	Time (s)	Calls	Avg (ms)
3*NSMs,	512	27	2048	67	1,58618	10	158,62

A primeira abordagem, e a mais simples, foi procurar maximizar a utilização dos recursos de processamento ou seja, aumentar a taxa de ocupação dos multiprocessadores e daí verificar se obteria melhor desempenho – ou seja, um menor tempo de execução. As configurações ensaiadas constam da Tabela 3.5 e verificou-se que a configuração de 16 *threads* por bloco foi a que menores tempos de execução teve apesar de apenas possuir uma taxa de ocupação de apenas 17%. Como se pode verificar a memória partilhada não constitui um fator limitativo da taxa de ocupação sendo apenas neste caso o número de registos utilizados o fator fixo realizados nos ensaios.

Tabela 3.5: Ensaio de configuração do *kernel* Update

#Block	#Threads/B	Reg	ShMem (B)	Occupancy (%)	Time (s)	Calls	Avg (ms)	Speedup
1*NSMs,	1024	27	4096	67	1,60583	10	160,58	-1,2%
3*NSMs,	512	27	2048	67	1,58618	10	158,62	ORIGINAL
6*NSMs,	256	27	1024	67	1,58361	10	158,36	0,2%
12*NSMs,	128	27	512	67	1,60136	10	160,14	-1,0%
24*NSMs,	64	27	256	33	1,42239	10	142,24	10,3%
48*NSMs,	32	27	128	17	0,887	10	88,7	44,1%
48	32	27	128	17	1,02869	10	102,87	35,1%
96*NSMs	16	27	64	17	0,76339	10	76,339	51,9%
96	16	27	64	17	0,76257	10	76,257	51,9%
384	4	27	16	17	1,33176	10	133,18	16,0%
384*NSMs	4	27	16	17	1,08001	9	120	31,9%

O passo seguinte foi verificar se limitando o número de registos para 20 (valor máximo que possibilita a taxa de ocupação atingir os 100%) se obteriam menores tempos de execução, o que pode ser observado na Tabela 3.6. Com a utilização da *flag* de compilação `maxrregcount=N` ou do qualificador `__launch_bounds__` observou-se que os menores tempos de execução se mantiveram na configuração de 16 *threads* por bloco e que, de uma forma geral, não ocorreram melhorias significativas com esta alteração.

Tabela 3.6: Ensaio de execução fixando o número de registos

#Block	#Threads/B	Reg	ShMem (B)	Occupancy (%)	Time (s)	Calls	Avg (ms)	Speedup
3*NSMs,	512	27	2048	67	1,58618	10	158,62	ORIGINAL
3*NSMs,	512	20	2048	100	1,62092	10	162,09	-2,2%
48*NSMs,	32	20	128	17	0,88175	10	88,175	44,4%
96*NSMs	16	20	64	17	0,77832	10	77,832	50,9%

O próximo passo foi verificar se particionando a memória L1 de forma a privilegiar a área dedicada a *cache* (reduzindo a área de memória partilhada) se obteriam melhores desempenhos. Para tal, utilizando a instrução `cudaFuncSetCacheConfig(GlobalSol, cudaFuncCachePreferL1)`, configurou-se a memória L1 com 48KB para *cache* e 16KB para memória partilhada. Os resultados obtidos são apresentados na Tabela 3.7 e verifica-se uma melhoria significativa no desempenho com esta funcionalidade ativa; assim, não havendo mais opções de configuração, estas são as melhores: reduzir o número de *threads* por bloco e aumentar a dimensão de *cache* da memória L1.

Tabela 3.7: Ensaio de execução fixando a preferência de modo Cache para a memória L1

#Block	#Threads/B	Reg	ShMem (B)	Occupancy (%)	Time (s)	Calls	Avg (ms)	Speedup
3*NSMs,	512	27	2048	67	1,58618	10	158,62	ORIGINAL
3*NSMs,	512	27	2048	67	1,49441	10	149,44	5,8%
48*NSMs,	32	27	128	17	0,51009	10	51,009	67,8%
96*NSMs	16	27	64	17	0,51236	10	51,236	67,7%

Numa análise à implementação pode-se observar que são efetuados vários acessos de leitura (linhas 12, 14, 20) e de escrita (linhas 15 e 21) não coalescentes e sobre a memória global, são usados mecanismos de sincronização de *threads* (linhas 28 e 36) e operações de escrita atómica, e utilizam-se várias instruções de controlo de fluxo de execução que podem originar alguma divergência.

```
1.  __global__ void Update(REAL *dev_displacements, REAL *dev_multipliers, REAL *dev_loc,
2.                 int *contactslist, REAL *radius, int nrcontacts, REAL *dev_error)
3.  {
4.    int i,j,id;
5.    extern __shared__ REAL buffer[];
6.    REAL tmp;
7.
8.    buffer[threadIdx.x]=0.0;
9.
10.   i=blockIdx.x*blockDim.x+threadIdx.x;
11.   while (i<nrcontacts) {
12.     id=contactslist[2*i]-1;
13.     for (j=0; j<6; j++)      {
14. tmp=dev_loc[12*i+j]-dev_displacements[6*id+j];
15. dev_multipliers[12*i+j]+=tmp*dev_penalty;
16. buffer[threadIdx.x]+=tmp*tmp;
17.       }
18.     id=contactslist[2*i+1]-1;
19.     for (j=0; j<6; j++)      {
20. tmp=dev_loc[12*i+j+6]-dev_displacements[6*id+j];
21. dev_multipliers[12*i+j+6]+=tmp*dev_penalty;
22. buffer[threadIdx.x]+=tmp*tmp;
23.       }
24.     i+=blockDim.x * gridDim.x;
25.   }
26.
27.   i = blockDim.x;
28.   __syncthreads();
29.   if (threadIdx.x==0 && blockDim.x%2)
30.     buffer[0] += buffer[i-1];
31.   while (i >1)
32.     {
33.       i /= 2;
34.       if (threadIdx.x < i)
35. buffer[threadIdx.x]+=buffer[threadIdx.x + i];
36.       __syncthreads();
37.     }
38.   if (threadIdx.x == 0)
39.     atomicAdd(dev_error, buffer[0]);
40. }
```

Relativamente aos acessos não coalescentes, tanto de leitura como de escrita, não se poderá agrupar os dados em virtude dos padrões de acesso serem completamente díspares uns dos outros (acessos por apontadores) e não ser comportável reservar memória partilhada para as dimensões envolvidas. Também não existe possibilidade de reduzir o número de registos pois as variáveis internas são de controlo à execução do *thread* e os registos utilizados estão implícitos no tipo de instruções utilizadas.

Finaliza-se então a otimização deste *kernel* com um ganho de 67,7% face ao *kernel* original, resultado da alteração do número de blocos e *threads* por bloco, e dimensionamento da *cache* da memória L1 para 48KB.

3.3.2 *Kernel* GlobalSol

Tal como o anterior, este *kernel* apresenta baixa eficiência nas operações de *store* e *load* sobre a memória global e uma eficiência de 73% da execução dos *warps*, o que revela um certo grau de divergência e uma taxa de acessos à memória global muito alta. Seguiu-se a mesma metodologia anterior de forma a atingir melhores desempenhos, ou seja, ensaiaram-se diversas configurações variando o número de blocos e de threads por bloco, depois observaram-se os efeitos sobre o desempenho por limitação do número de registos, e em seguida com a configuração do espaço da memória L1 para utilização de 48KB para *cache*.

Os valores obtidos do ensaio e constantes na Tabela 3.8 mostram que os menores tempos de execução são obtidos quando o *kernel* possui uma configuração de 16 *threads* por bloco.

Tabela 3.8: Ensaio de configurações realizadas ao número de blocos e *threads*

#Block	#Threads/B	Reg	ShMem (B)	Occupancy (%)	Time (s)	Calls	Avg (ms)	Speedup
1*NSMs	1024	23	4100	67	1,06624	9	118,47	1,4%
2*NSMs	672	23	2692	88	1,0815	9	120,17	0,0%
3*NSMs	512	23	2052	67	1,20959	9	134,4	-11,8%
3*NSMs	448	23	1796	88	1,08169	9	120,19	ORIGINAL
8*NSMs	192	23	772	88	1,15723	9	128,58	-7,0%
12*NSMs	128	23	516	67	1,16155	9	129,06	-7,4%
24*NSMs	64	23	260	33	1,03095	9	114,55	4,7%
48*NSMs	32	23	132	17	0,94816	9	105,35	12,3%
96*NSMs	16	23	68	17	0,8448	9	93,867	21,9%

A avaliação realizada quando se fixou o número de registos para 20, pode ser visualizada na Tabela 3.9 e constata-se que não há ganhos com esta alteração.

Tabela 3.9:Resultados obtidos ao fixar o número de registos

#Block	#Threads/B	Reg	ShMem (B)	Occupancy (%)	Time (s)	Calls	Avg (ms)	Speedup
3*NSMs	448	23	1796	88	1,08169	9	120,19	ORIGINAL
3*NSMs	448	20	1796	88	1,08965	9	121,07	-0,7%
48*NSMs,	32	20	132	17	0,94771	9	105,3	12,4%
96*NSMs	16	20	68	17	0,85143	9	94,603	21,3%

De seguida apresenta-se na Tabela 3.10 os valores obtidos quando se configurou a memória L1 com 48KB de *cache*, podendo-se constatar um ligeiro ganho relativamente aos resultados obtidos anteriormente.

Tabela 3.10: Resultados obtidos quando a memória L1 está configurada a 48KB

#Block	#Threads/B	Reg	ShMem (B)	Occupancy (%)	Time (s)	Calls	Avg (ms)	Speedup
3*NSMs	448	23	1796	88	1,08169	9	120,19	ORIGINAL
3*NSMs	448	23	1796	88	1,05142	9	116,82	2,8%
48*NSMs,	32	23	132	17	0,85439	9	94,932	21,0%
96*NSMs	16	23	68	17	0,79621	9	88,467	26,4%

Numa avaliação mais cuidadosa à implementação abaixo pode-se observar a existência de um mecanismo de sincronização, `__syncthreads()` (`linha 11`), que obriga a que todos os

threads de um bloco aguardem até que todos completarem a sua execução. Pode-se também constatar que, apesar de se copiar os dados da memória global para a memória partilhada (linha 9) existem outras operações de leitura (linhas 17, 19 e 21) e uma operação de escrita (linha 21) em memória global. Pode-se também verificar que ocorrem leituras encadeadas cujos seus *outputs* são *inputs* de outras funções. Por fim, a inclusão de instruções *if* e *for* pode originar algum grau de divergência.

```
1.  __global__ void GlobalSol(int nrparticles, int *dev_Cxadj, int *dev_Cadjncy,
2.                                                  REAL *dev_displacements, REAL *dev_forces,
3.                                                  REAL *dev_loc, REAL *dev_diag_mass){
4.    extern __shared__ int xadj[];
5.    int i,j,k;
6.    REAL tmp, coef;
7.
8.    for (i=blockIdx.x*blockDim.x+threadIdx.x; i<nrparticles; i+=blockDim.x * gridDim.x) {
9.      xadj[threadIdx.x+1]=dev_Cxadj[i+1];
10.     if (threadIdx.x==0) xadj[threadIdx.x]=dev_Cxadj[i];
11.     __syncthreads();
12.
13.     coef=dev_penalty*(xadj[threadIdx.x+1]-xadj[threadIdx.x]);
14.
15. #pragma unroll
16.     for (k=0; k<6; k++){
17.       tmp=dev_forces[6*i+k];
18.       for (j=xadj[threadIdx.x]; j<xadj[threadIdx.x+1]; j++)
19.         tmp+=dev_loc[dev_Cadjncy[j]+k];
20.
21.       dev_displacements[6*i+k]=tmp/(coef+dev_diag_mass[6*i+k]);
22.     }
23.   }
24. }
```

Seria desejável reformular a implementação no sentido de procurar reduzir o número de variáveis (logo, registos), eliminar o mecanismo de sincronização e, se possível, minimizar o número de operações de leitura da memória global. Assim, fizeram-se pequenas alterações ao código no sentido de remover o mecanismo de sincronização e reformular o acesso aos dados (Anexo B):

1. Leitura direta sobre a memória global e sem mecanismos de sincronização;
2. Leitura de dados para duas variáveis locais e sem mecanismos de sincronização;
3. Leitura de dados para variáveis na memória partilhada e sem mecanismos de sincronização.

Os resultados obtidos constam da Tabela 3.11 e mostram que em geral não se obtém melhor *speedup* e que os poucos casos que apresentaram menor tempo de execução não são significativos.

Tabela 3.11: Resultado dos ensaios realizados com diferentes acessos à memória e sem mecanismos de sincronização

#Block	#Threads/B	Reg	ShMem (B)	Occupancy (%)	Time (s)	Calls	Avg (ms)	Speedup	Obs.
3*NSMs	448	23	1796	88	1,08169	9	120,19		ORIGINAL
3*NSMs	448	21	0	88	1,08956	9	121,06	-0,7%	leitura direta à mem global
3*NSMs	448	21	0	88	1,08742	9	120,83	-0,5%	leitura para registos
3*NSMs	448	22	3588	88	1,08973	9	121,08	-0,7%	leitura para mem partilhada
48*NSMs	32	23	132	17	0,94816	9	105,35	12,3%	Ref base
48*NSMs,	32	21	0	17	0,95064	9	105,63	12,1%	leitura direta à mem global
48*NSMs,	32	21	0	17	0,94648	9	105,16	12,5%	leitura para registos
48*NSMs,	32	22	260	17	0,94998	9	105,55	12,2%	leitura para mem partilhada
96*NSMs	16	23	68	17	0,8448	9	93,867	21,9%	Ref base
96*NSMs	16	21	68	17	0,85424	9	94,916	21,0%	leitura direta à mem global
96*NSMs	16	21	0	17	0,84421	9	93,801	22,0%	leitura para registos
96*NSMs	16	22	132	17	0,85194	9	94,659	21,2%	leitura para mem partilhada

Considerou-se ainda a possibilidade de alterar a implementação atual, na qual se usa um número de fixos de blocos cujas *threads*, com um *stride* definido, percorrem toda a estrutura de dados de partículas. Na nova implementação usou-se uma *thread* por partícula (dentro dos limites das capacidades da GPU). Assim, para o caso em estudo com a dimensão de 1.088.098 partículas foram realizados ensaios para os blocos com melhor desempenho; os resultados obtidos podem ser visualizados na Tabela 3.12 onde se pode constatar que os ganhos obtidos são semelhantes à implementação já existente, apesar de se ter conseguido baixar o número de registos, de tentar uma taxa de ocupação de 100% dos multiprocessadores, utilizando os diferentes níveis de memória.

Tabela 3.12: Ensaios realizados com novo *kernel GlobalSol*

#Block	#Threads/B	Reg	ShMem (B)	Occupancy (%)	Time (s)	Calls	Avg (ms)	Speedup	Obs.
3*NSMs	448	23	1796	88	1,08169	9	120,19		ORIGINAL
2430	448	20	0	88	1,07739	9	119,71	0,4%	novo *kernel* + registos
5668	192	20	0	100	1,08474	9	120,53	-0,3%	novo *kernel* + registos
34001	32	20	132	17	0,9213	9	102,37	14,8%	novo *kernel* + shared mem
34001	32	20	0	17	0,92094	9	102,33	14,9%	novo *kernel* + registos
34001	32	21	0	17	0,92362	9	102,62	14,6%	novo *kernel* com acesso direto à Global Mem
34001	32	20	0	17	0,85491	9	94,99	21,0%	novo *kernel* + registos + cudaFuncCachePreferL1

3.3.3 *Kernel* CalcA

Este *kernel* apresentou-se com uma baixa taxa de ocupação, baixa eficiência nas operações sobre memória global, e algum grau de divergência na execução dos *warps*; o número de registos e a quantidade de memória partilhada requerida estão no máximo (ou perto) dos valores admissíveis para a GPU C2050.

Seguindo a mesma metodologia, ensaiaram-se um conjunto de configurações das quais se obtiveram os valores apresentados na Tabela 3.13, na qual se pode ainda verificar que a configuração ideal é de 384 *threads* por bloco.

Tabela 3.13: Ensaio realizado com diferentes configurações

#Block	#Threads/B	Reg	ShMem (B)	Occupancy (%)	Time (s)	Calls	Avg (ms)	Speedup
NSMs	512	63	43008	33	1,76214	20	88,107	ORIGINAL
3*NSMs,	512	63	43008	33	1,66148	20	83,074	5,7%
4*NSMs,	384	63	32256	25	1,39897	20	69,949	20,6%
6*NSMs,	256	63	21504	33	1,68828	20	84,414	4,2%
8*NSMs,	192	63	16128	25	1,41272	20	70,636	19,8%
12*NSMs,	128	63	10752	33	1,71494	20	85,747	2,7%
16*NSMs,	96	63	8064	31	1,808	20	90,4	-2,6%
24*NSMs,	64	63	5376	33	1,73559	20	86,78	1,5%
48*NSMs,	32	63	2688	17	1,63852	20	81,926	7,0%
96*NSMs	16	63	1344	17	1,64518	20	82,259	6,6%

Atendendo ao número elevado de registos que o *kernel* necessita não seria coerente testar configurações que os limitem ainda mais, pois a arquitetura está programada para fazer o *spill* dos dados para a memória local. No entanto, por uma questão de confirmação e completude, apresenta-se na Tabela 3.14 o resultado obtido para 20 registos que, como se esperava é (cerca de 21%) pior do que a versão original.

Tabela 3.14: Ensaio realizado limitando o número de registos a 20

#Block	#Threads/B	Reg	ShMem (B)	Occupancy (%)	Time (s)	Calls	Avg (ms)	Speedup	Obs.
NSMs	512	63	43008	33	1,76214	20	88,107		ORIGINAL
4*NSMs,	384	63	32256	25	1,39897	20	69,949	20,6%	
4*NSMs,	384	20	32256	25	2,13558	20	106,78	-21,2%	limitado a 20 registos

Verificou-se igualmente que com uma configuração para 48KB de *cache* da memória L1, os resultados obtidos (Tabela 3.15) mostram claramente que não existe benefício.

Tabela 3.15: Ensaio realizado com configuração de 48KB de *cache* para a memória L1

#Block	#Threads/B	Reg	ShMem (B)	Occupancy (%)	Time (s)	Calls	Avg (ms)	Speedup
NSMs	512	63	43008	33	1,76214	20	88,107	ORIGINAL
8*NSMs,	192	63	16128	13	1,78103	20	89,052	-1,1%
48*NSMs	32	63	2688	17	1,79895	20	89,948	-2,1%

Feita a análise detalhada ao *kernel* verificou-se que o tipo de operações utilizadas são operações aritméticas simples, do tipo matricial e vetorial e ainda de leitura e escrita em memória global. Assim, procurou-se utilizar as bibliotecas otimizadas pela NVIDIA para cálculo vetorial e matricial, *cuBLAS* e *cuSparse*, de forma a verificar se conseguiriam ainda melhores resultados (a implementação pode ser visualizada no Anexo B). Constatou-se que, não, pois não só o tempo de execução aumentou cerca de 11,8%, como também houve um acréscimo de 76,6% no consumo de memória global.

```
Total simulation time= 0 h 12 m 45.8610 s    MEM = 886 MB  (VERSÃO ORIGINAL)
31.72%  231.512s      6000  38.585ms  37.861ms  39.263ms  CalcA(float*, float*, float*, float*, int*, float*, int)

Total simulation time= 0 h 14 m 16.6591 s    MEM = 1565 MB (CUBLAS && CUSPARSE)
 7.91%  65.0994s      6000  10.850ms  10.617ms  11.923ms  CalcA1(float*, float*, float*, int*, float*, int)
51.26%  421.991s      6000  70.332ms  43.108ms  97.639ms  void csrMv_ci_kernel<float, int=7, int=1, int=5>(cusparseCsrMvParams<float>, int, int)
```

Figura 3.11: Tempo de execução com utilização das bibliotecas cuBLAS e CUSPARSE

3.3.4 *Kernel* Local2Global

À semelhança do *kernel* anterior também este possui o número de registos e quantidade de memória demasiado elevados para que se consiga atingir uma boa taxa de ocupação dos

multiprocessadores e, por outro lado, possui também baixa eficiência nas operações de acesso à memória global.

Novamente seguindo a metodologia anterior, procurou-se encontrar a melhor configuração do *kernel* que reduz o tempo de execução; os ensaios realizados e resultados conseguidos são apresentados na Tabela 3.16, onde se pode verificar as duas melhores configurações são as de 384 e 192 *threads* por bloco.

Tabela 3.16: Ensaio de configurações do *kernel* Local2Global

#Block	#Threads/B	Reg	ShMem (B)	Occupancy (%)	Time (ms)	Calls	Avg (ms)	Speedup
1*NSMs,	512	63	43008	33	352,150	10	35,215	ORIGINAL
4*NSMs,	384	63	32256	25	286,120	10	28,612	18,8%
6*NSMs,	256	63	21504	33	343,330	10	34,333	2,5%
8*NSMs,	192	63	16128	25	287,500	10	28,75	18,4%
12*NSMs,	128	63	10752	33	345,890	10	34,589	1,8%
16*NSMs,	96	63	8064	31	342,130	10	34,213	2,8%
24*NSMs,	64	63	5376	33	348,600	10	34,86	1,0%
48*NSMs,	32	63	2688	17	311,920	10	31,192	11,4%

Na Tabela 3.17 pode observar-se o impacto no tempo de execução quando se limitou o número de registos para 20: foi muito penalizador, o que apenas confirmou o que já se esperava, pois o número de variáveis (e consequentemente de registos) presentes no código é muito grande.

Tabela 3.17: Ensaio limitando o número de registos

#Block	#Threads/B	Reg	ShMem (B)	Occupancy (%)	Time (ms)	Calls	Avg (ms)	Speedup
1*NSMs,	512	63	43008	33	352,150	10	35,215	ORIGINAL
4*NSMs,	384	20	32256	25	636,150	10	63,615	-80,6%
48*NSMs	32	20	2688		607,430	10	60,743	-72,5%

Relativamente à configuração da memória L1 para trabalhar preferencialmente como *cache*, não houve ganhos significativos (Tabela 3.18).

Tabela 3.18: Ensaio com configuração da memória L1

#Block	#Threads/B	Reg	ShMem (B)	Occupancy (%)	Time (ms)	Calls	Avg (ms)	Speedup
1*NSMs,	512	63	43008	33	352,150	10	35,215	ORIGINAL
8*NSMs,	192	63	16128	25	286,530	10	28,653	18,6%
48*NSMs,	32	63	2688	25	355,700	10	35,57	-1,0%

Feita a análise detalhada ao código do *kernel* pode verificar-se que, tal como o anterior, este efetua muitas operações aritméticas simples, do tipo matricial e vetorial, e executa várias operações de leitura e escrita à memória global. Nesse sentido, procurou-se utilizar as bibliotecas *cuBLAS* e a *cuSparse* de forma a verificar se conseguíamos obter ganhos efetivos. Contudo, tal como sucedeu no *kernel CalcA*, constatou-se um acréscimo no tempo de

execução e no consumo de memória.

3.3.5 *Kernel* LocalMin

Este *kernel* apresentou-se com baixa taxa de ocupação dos multiprocessadores, baixa eficiência das operações de *load* e *store* sobre a memória global e um grau significativo de divergência na execução dos *warps*.

Seguindo a mesma metodologia, apresentam-se na Tabela 3.19 os resultados obtidos para as várias configurações ensaiadas, sendo que a melhor, embora por muito pouco, é a de 704 *threads* por bloco.

Tabela 3.19: Ensaio de diferentes configurações do *kernel*

#Block	#Threads/B	Reg	ShMem (B)	Occupancy (%)	Time (ms)	Calls	Avg (ms)	Speedup
2*NSMs	704	44	42240	46	172,09	10	17,209	7,9%
3*NSMs	512	44	30720	33	186,11	10	18,611	0,4%
2*NSMs,	384	44	23040	25	186,79	10	18,679	ORIGINAL
6*NSMs,	256	44	15360	33	184,96	10	18,496	1,0%
8*NSMs,	192	44	11520	25	191,55	10	19,155	-2,5%
12*NSMs,	128	44	7680	33	192,55	10	19,255	-3,1%
16*NSMs,	96	44	5760	31	199,02	10	19,902	-6,5%
24*NSMs,	64	44	3840	33	185,67	10	18,567	0,6%
48*NSMs,	32	44	1920	17	295,91	10	29,591	-58,4%

Limitando o número de registos a 20, os resultados, registados na Tabela 3.20, mostram de novo que o *spill* de registos para memória local é bastante penalizador para o desempenho.

Tabela 3.20: Ensaio limitando o número de registos a 20

#Block	#Threads/B	Reg	ShMem (B)	Occupancy (%)	Time (ms)	Calls	Avg (ms)	Speedup
2*NSMs	384	44	23040	25	186,79	10	18,679	ORIGINAL
2*NSMs	384	20	23040	25	324,42	10	32,442	-73,7%
2*NSMs	704	20	42240	46	315,59	10	31,559	-69,0%

A Tabela 3.21 colige os ensaios realizados, após configurarmos a memória L1 como uma *cache* de 48KB; verificamos que o desempenho deste *kernel* piora relativamente ao original, e não se altera significativamente (piorou 0,2%) relativamente à melhor configuração obtida anteriormente.

Tabela 3.21: Ensaio com a configuração da memória L1 para 48KB de *cache*

#Block	#Threads/B	Reg	ShMem (B)	Occupancy (%)	Time (ms)	Calls	Avg (ms)	Speedup
2*NSMs	384	44	23040	25	186,79	10	18,679	ORIGINAL
2*NSMs	384	44	23040	25	218,03	10	21,803	-16,7%
2*NSMs,	704	44	42240	46	172,41	10	17,241	7,7%

Analisado o *kernel* com algum detalhe pode-se observar que se trata de um *kernel* simples que, através de um ciclo, faz o *load* de dados (não coalescentes) da memória global

para a memória partilhada, executa várias operações aritméticas, com algum ênfase para potências e raízes quadradas (que poderão ser sujeitas a otimizações) e volta a fazer o *store* dos dados (não coalescente) para memória global. O elevado número de *threads* inativos que são relatados no *profile* inicial deve-se a divergências que resultam das várias instruções de controlo de fluxo que o *kernel* possui. Não nos parece existirem outras otimizações para além da que usamos: uma *flag* de compilação que obriga ao compilador converter as funções *sqrt*, *pow*, *x/y* em funções otimizadas, à custa de alguma perda de precisão. Os ensaios realizados demonstraram que não houve ganhos relativamente aos resultados anteriores.

Tabela 3.22: Ensaio com a simplificação de instruções utilizadas

#Block	#Threads/B	Reg	ShMem (B)	Occupancy (%)	Time (ms)	Calls	Avg (ms)	Speedup
2*NSMs,	384	44	23040	25	186,79	10	18,679	ORIGINAL
2*NSMs,	384	44	23040	25	198,98	10	19,898	-6,5%
2*NSMs,	704	44	42240	46	176,83	10	17,683	5,3%

3.3.6 Kernels especializados

Relativamente aos *kernels* especializados e restantes *kernels* da aplicação, foi seguida a mesma metodologia ficando remetido para anexo o trabalho realizado em virtude de não existirem grandes alterações relativamente aos *kernels* principais (Anexo C).

3.3.7 A aplicação Sphaerae

Tendo em consideração os dados da avaliação inicial da aplicação, poder-se-á obter melhor desempenho caso sejam consideradas as otimizações relativas à concorrência de operações (transferência de dados e execução de *kernels*) e à utilização de memória não paginável.

A concorrência de operações, explorada na execução de *kernels*, foi realizada com recurso à utilização de *streams* e operações assíncronas. Para tal, foram identificados os *kernels* que poderiam beneficiar da utilização de mecanismos de concorrência ficando organizados conforme se mostra na Figura 3.12[13].

[13] Aplica-se a mesma organização aos *kernels* especializados ou seja, onde se lê na Figura 3.12, CalcA passa a ler-se CalcA2 e CalcAWall passa a CalcAwall2.

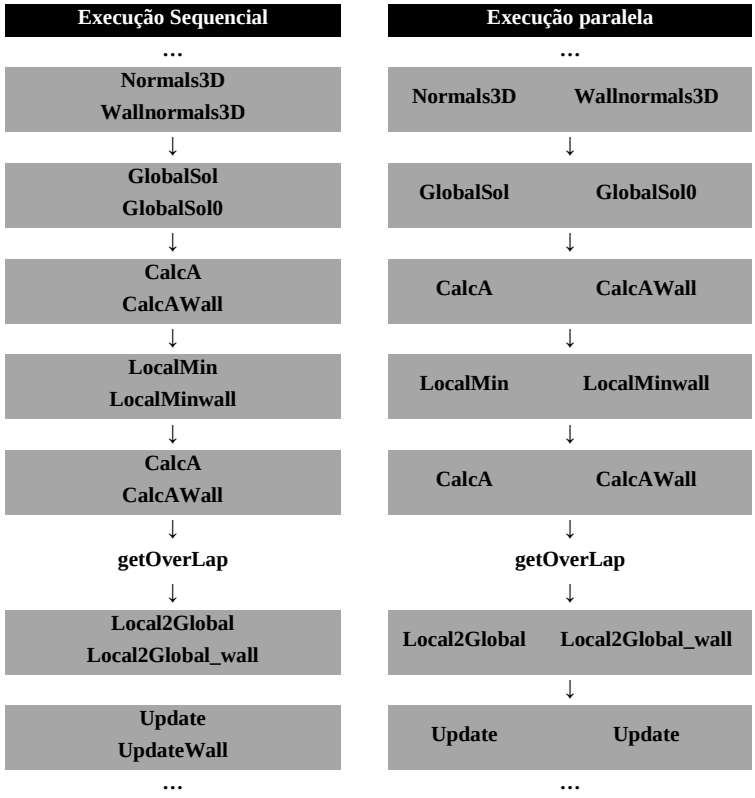


Figura 3.12: Organização dos *kernels* para usufruir dos mecanismos de concorrência

Para introduzir a concorrência foi necessário incluir o seguinte código na implementação:

```
1.  //Create streams
2.  EXTERN cudaStream_t stream0;
3.  EXTERN cudaStream_t stream1;
4.
5.  ...
6.
7.  //kernels definition
8.  GlobalSol<<<#Blocks, #Threads, #SharedMem,stream0>>>
9.  GlobalSol0<<<#Blocks, #Threads, #SharedMem,stream1>>>
10. ...
11.
12. //Destroy streams
13. cudaStreamDestroy(stream0);
14. cudaStreamDestroy(stream1);
```

As cópias assíncronas de dados entre o *host* e o *device* não foram tentadas uma vez que a implementação atual está organizada de forma a que a maior parte dos dados sejam copiados para o *device* antes do início da execução na GPU e as transferências entre esta e o *host*, durante a execução, são residuais.

De uma forma mais transversal procurou-se ainda aumentar o *throughput* de execução de instruções através de *flags* de compilação que geram instruções cuja execução envolve menor número de ciclos de relógio. As *flags* de compilação utilizadas foram:

- *-use_fast_math*, usa “instruções matemáticas” mais simples;
- *-ftz = false*, números não normalizados são convertidos em zeros;
- *-prec div=false*, menor precisão nas operações de divisão;
- *-prec-sqrt=false*, menor precisão nas operações de raízes quadradas;

A utilização destas *flags* acarreta alguma perda de precisão que foi analisada no *output* gerado pelo simulador, mas que se revelou nulo nos ensaios realizados. Explorou-se também, ainda que pontualmente, a substituição de operações por outras equivalentes, como por exemplo potências por operações de multiplicação, mas concluímos que, implementadas estas otimizações, não ocorreram melhorias no desempenho.

4

4 Avaliação

4.1 Instrumentos de avaliação e validação

As expectativas iniciais consistiam em reduzir substancialmente o tempo de simulação e se possível, aumentar o número de partículas. Para atingir esses objetivos foi definida uma estratégia baseada num processo iterativo com os seguintes passos: realizar avaliações de desempenho da implementação (usando ferramentas de *profiling)*, identificar "módulos" a otimizar, realizar as otimizações, avaliar o progresso e retomar se necessário.

Realizadas as otimizações foi necessário avaliar e validar os resultados e para tal, tomaram-se como valores de referência, os tempos de execução obtidos com a implementação original bem como os seus *outputs* gerados - por exemplo, os valores de erro, fator energia, fator de penalização e valor máximo de sobreposição. Estes *logs* são gerados em zonas predefinidas para cada passo e permitem fazer o *tracking* da simulação e a comparação com os *logs* gerados noutras simulações. Determinou-se também a grandeza do erro associado aos resultados obtidos através do somatório dos quadrados das diferenças entre os valores originais e os obtidos após realizar as otimizações. Verificou-se que a grandeza do erro foi nula resultado de se tratar de processos completamente determinísticos e que a perda de precisão de eventuais operações não tiveram qualquer impacto.

O conjunto de dados de entrada da simulação é composto por dois ficheiros, sendo um referente às partículas e outro referente à superfície que limita essas partículas. O ficheiro de partículas tem três "partes": parâmetros da aplicação, dados de posição e dados de forças relativos a cada uma das partículas a avaliar. Relativamente aos parâmetros da aplicação foram considerados, para efeitos de avaliação, o atrito entre partículas (*mu*), o atrito partícula-

superfície (*muw*) e a resistência ao rolamento (*mur*)[14].

```
# vtk DataFile Version 4.2
3D  time=0.500000D-01  theta=0.100000D+01  steps=1      g=0.981000D+01  rho=0.100000D+01
jcoef=0.100000D+01 mu=0.500000D+00 muw=0.500000D+00 mur=0.100000D+00    tol=0.100000D-15
maxiter=3000 r=0.100000D+03
BINARY
DATASET UNSTRUCTURED_GRID

POINTS     6000  floaT
…
POINT_DATA     6000
SCALARS radius double
LOOKUP_TABLE default
..
VECTORS velocity double
…
VECTORS ang_velocity double
…
```

Para obter o *profiling* da aplicação foram utilizados conjuntos de partículas com dimensões variadas; contudo, a avaliação final dos resultados foi sempre realizada com 1 milhão de partículas. Foram igualmente consideradas dois tipos de limites de superfície, uma mais densa que outra (maior número de partículas) por forma a melhor avaliar e execução concorrente dos *kernels*.

Relativamente ao *hardware* utilizado para a avaliação foram usados os GP-GPUs NVIDIA Tesla™ C2050 e Tesla™ M2075, cujas características constam da Tabela 4.1.

Tabela 4.1: Dispositivos utilizados na avaliação

	Tesla M2075	Tesla C2050
CUDA Driver Version / *Runtime* Version	5.5 / 5.5	6.5 / 6.5
CUDA Capability	2.0	2.0
Total amount of global memory:	5375 MBytes	3072 MBytes
CUDA *Cores*	(14) SM x (32) CUDA *Cores* = 448	(14) SM x (32) CUDA *Cores* = 448
GPU Clock rate	1147 MHz (1.15 GHz)	1147 MHz (1.15 GHz)
Memory Clock rate	1566 Mhz	1500 Mhz
Memory *Bus* Width	384-bit	384-bit
L2 Cache Size	786432 bytes	786432 bytes
Total amount of constant memory	65536 bytes	65536 bytes
Total amount of shared memory per block	49152 bytes	49152 bytes
Total number of registers available per block	32768	32768
Warp size	32	32
Maximum number of *threads* per multiprocessor	1536	1536
Maximum number of *threads* per block	1024	1024
Maximum memory pitch	2147483647 bytes	2147483647 bytes
Concurrent copy and *kernel* execution	Yes with 2 copy engine(s)	Yes with 2 copy engine(s)
Support *host* page-locked memory mapping	Yes	Yes
Device PCI *Bus* ID / PCI location ID	3 / 0	3 / 0

[14] Os restantes parâmetros mantiveram-se constantes.

4.2 Avaliação comparativa

Realizadas as otimizações descritas no capítulo anterior, interessa agora apresentar um quadro resumo, na Tabela 4.2, com as otimizações realizadas e os ganhos verificados relativamente à implementação original.

Tabela 4.2: Quadro resumo das otimizações conseguidas

KERNEL	ALTERAÇÃO block && *threads*	ALTERAÇÃO Configuração mem L1	CONCORRÊNCIA	INTRODUÇÃO Novo *kernel* /biblioteca	SPEEDUP
GlobalSol	✓	✓	✓	✓	26,4%
GlobalSol0	✓		✓		33,1%
Update	✓	✓	✓		67,8%
UpdateWall	✓	✓	✓		51,0%
CalcA	✓		✓		20,6%
CalcAwall	✓	✓	✓		25,2%
CalcA2	✓		✓		19,5%
CalcA2wall	✓	✓	✓		43,5%
Local2Global	✓		✓		18,8%
Local2Global_wall	✓		✓		1,7%
Local2Global2	✓		✓		12,5%
Local2Global2_wall	✓		✓		4,2%
LocalMin	✓		✓		7,9%
LocalMinWall	✓		✓		15,6%
LocalMin0	✓		✓		0,1%
LocalMin0wall	✓	✓	✓		15,6%
LocalMin1	✓	✓	✓		23,7%
LocalMin1Wall	✓	✓	✓		4,7%
LocalRHS					0,0%
getOverlap		✓			12,6%
getEnergy		✓			20,9%
normals3D	✓		✓		15,5%
wallnormals3D	✓		✓		5,5%

Escolhidas as melhores configurações para cada *kernel* ensaiamos a aplicação para 3 variantes das interações possíveis, tendo obtido os ganhos expressos na Tabela 4.3. Em resumo, **o trabalho realizado traduziu-se num *speedup* de 32%**.

Tabela 4.3: Comparativos de tempos de execução

# PARTÍCULAS	#PASSOS	#MAX ITER	TOL	ATRITO P-P	ATRITO P-S	RES. ROLAM	ORIGINAL	OTIMIZADO	SPEEDUP
1M	1	3000	0,10D-15	✓	✓	✓	29,27’	20,42’	30%
1M	1	3000	0,10D-15	✓	✓		30,87’	23,67’	23%
1M	1	3000	0,10D-15	✓			30,90’	23,71’	23%
1M	**25**	3000	0,10D-15	✓	✓	✓	635,02’	431,07’	**32%**

4.3 Análise e discussão dos resultados

Tentaremos agora justificar e/ou comprovar os ganhos obtidos com a exibição de novos dados de *profiling* que nos permitam comparar a aplicação no seu estado inicial com a versão

depois de otimizada.

Tabela 4.4: Padrão de utilização de memória e larguras de banda utilizada

INICIAL

		Update	*GlobalSol*	*CalcA*	*Local2Global*	*LocalMin*
L1 /Shared Memory	*Transactions*	178672601	147674674	145209934	96131100	47609084
	Bandwidth (GB/s)	134,475	158	194,526	306,662	246,509
	Utilization	Baixa	Baixa	Baixa	Média/Baixa	Média/Baixa
L2 Cache	*Transactions*	617525652	519384284	513850238	189054602	86435550
	Bandwidth (GB/s)	124,381	138,813	187,27	171,809	124,722
	Utilization	Média/Alta	Média/Alta	Alta/Max	Alta	Média/Alta
Device memory	*Transactions*	534013421	475762649	300486013	111639545	60319120
	Bandwidth (GB/s)	107,56	127,155	109,511	101,455	87,037
	Utilization	Alta	Alta/Máx	Alta	Alta	Média/alta

FINAL

		Update	*GlobalSol*	*CalcA*	*Local2Global*	*LocalMin*
L1 /Shared Memory	*Transactions*	167600293	147210976	145081621	96701499	48449425
	Bandwidth (GB/s)	392,49	193	241,577	406,508	309,883
	Utilization	Média/Baixa	Baixa	Média/Baixa	Média/Baixa	Média/Baixa
L2 Cache	*Transactions*	371971808	407604582	490915180	174253812	88658774
	Bandwidth (GB/s)	234,206	133,556	222,382	208,929	157,722
	Utilization	Max	Média/Alta	Max	Max	Alta
Device memory	*Transactions*	129045223	374232413	156520598	89273339	59046708
	Bandwidth (GB/s)	81,25	122,622	70,903	107,038	105,043
	Utilization	Média/Alta	Alta/Máx	Média	Alta	Alta

A Tabela 4.41 mostra claramente, na nossa opinião, a razão dos ganhos conseguidos: observamos que no geral (com apenas uma quebra de 4% no GlobalSol) os débitos das memórias L1 e L2 subiram apreciavelmente sendo (aproximadamente) o ganho mais pequeno conseguido para a L1 de 122%, no *GlobalSol*, e o maior de 291 %, no *Update*; já no nível L2, o menor ganho (de facto, foi uma perda) foi de -4%, no *GlobalSol*, e o maior de 188%, no *Update*. Se continuarmos a analisar os dados apresentados na figura, focando agora a nossa atenção na largura de banda conseguida nos acessos à memória global, vemos que salvo uma ou outra exceção pouco significativa, o número de transações executadas sobre cada um dos níveis de memória diminuiu, por vezes até bastante. Tal é consistente com a análise anterior, pois sabendo-se que os mesmos dados vão ter de forçosamente ser referenciados pelas computações, e que o número de transações sobre a memória global diminuiu, a razão só pode ser um melhor aproveitamento das *caches*.

Continuando a discussão dos resultados pós otimização resta-nos, para concluir, analisar de que forma as otimizações se refletem nas causas de "menor desempenho" apontadas pelo profiler quando se analisou a versão inicial da aplicação. Para isso exibimos na Figura 4.2 as causas de *stall* observadas na execução do *kernel Update* (sendo que os gráficos para os outros kernels são razoavelmente idênticos). Isto é, mostramos o que está a estagnar a execução da GPU e, logo, a diminuir o seu desempenho: do lado esquerdo apresentamos o diagrama correspondente à versão inicial da aplicação; do lado direito, o que corresponde à versão otimizada.

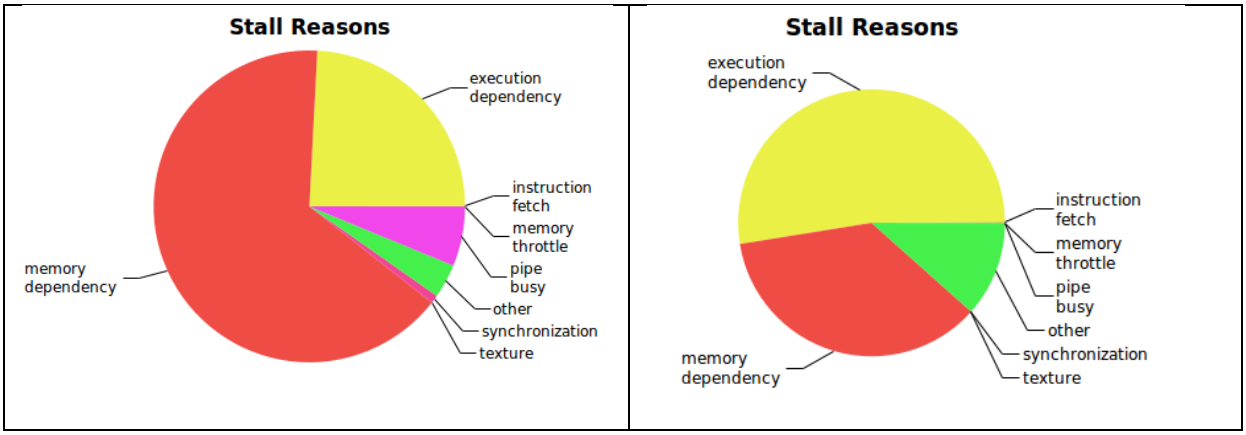


Figura 4.2: Razões para stall do kernel Update: à esquerda, a versão inicial; à direita, a otimizada.

Da observação da Figura 4.2 salta imediatamente à vista a diminuição para cerca de metade da latência de acesso à memória (a vermelho) como razão para travar o progresso das computações; agora, cresce a fatia correspondente à dependência de dados de execução como causa principal de *stall*. Ou seja, em termos simples, os ganhos que conseguimos acontecem porque reduzimos apreciavelmente uma causa de *bottleneck*, trocando-a por outra; mas, como o tempo durante o qual uma computação "emperrada" tem que esperar por um dado produzido por outra é tipicamente muito menor que o tempo de acesso à memória, a troca *stall*-por-dependência-de-memória por *stall*-por-dependência-de-computação acaba por ser positiva.

5

5 Conclusões

5.1 Balanço do trabalho

Tendo sido estabelecido como tema deste trabalho a "Otimização de uma implementação em GPUs de um algoritmo de simulação de materiais granulares" houve, no princípio uma certa apreensão sobre a capacidade de se atingirem efetivamente resultados significativos uma vez que tínhamos pela frente dois desafios totalmente novos: a tecnologia das GP-GPUs, cujo modelo de programação se afasta muito da típica programação sequencial; e o tema, que estando noutro domínio do conhecimento, é necessário compreender para entender o funcionamento da implementação existente.

Pesem embora as dúvidas inicias, o trabalho anteriormente efetuado na fase de preparação, visando nomeadamente o estudo e compreensão das arquiteturas GP-GPU, seus modelos de programação, ambientes de desenvolvimento e, em especial, da arquitetura e modelo de programação da tecnologia NVIDIA, bem como uma breve exploração da implementação a otimizar, permitiu que nesta segunda fase se obtivesse uma melhoria efetiva, na ordem dos 30%, da implementação já existente.

Tratando-se de uma otimização de um algoritmo na tecnologia NVIDIA, num modelo de programação que pode, numa certa perspetiva, ser designado de "baixo nível", realizou-se um trabalho exaustivo de *profiling*, de testes de configurações de *kernels*, de *fine tuning* dos recursos da GPU (registos, hierarquia de memórias – global, L2 e L1 – particionamento desta última entre *caching* e *sharing*) com o objetivo de obter resultados promissores que se traduzissem num melhor desempenho do algoritmo. Esgotado este estudo, abordou-se de forma minimalista (pois era especificamente "proibido" neste trabalho) a remodelação e reconstrução de partes do algoritmo, através de modificação direta do código ou recorrendo a bibliotecas otimizadas já existentes (o único caso em que se transformou algumas estruturas de dados noutras equivalentes para se poderem usar as funções de biblioteca). Sendo um processo iterativo (avaliar, identificar, otimizar e reavaliar) que prosseguia até se atingir um

grau satisfatório ou até ser impossível fazer melhor, permitiu tirar conclusões relativamente à implementação existente e a possíveis realizações de trabalhos futuros.

Assim sendo, ficou provado pelo *profiling* obtido que a implementação existente é *memory-bound*, o que coloca logo à partida algumas dificuldades neste tipo de arquiteturas, que têm uma capacidade muito limitada de armazenamento de dados nos diferentes níveis da hierarquia da memória (especialmente no nível L1). Não se conseguindo, portanto, aproveitar melhor as capacidades de *caching*, cai-se então numa cascata de "problemas": incapacidade de aproveitar as elevadas larguras de banda disponíveis nos diferentes *busses* (só o *bus* de acesso à memória global tem bom débito); latência no acesso aos dados e *stall* na execução de instruções consequência de dependência dos *inputs* necessários à execução, etc. Verificou-se ainda pelo *profiling* que a grande maioria das instruções aritméticas e lógicas geradas são instruções simples pelo que não prejudicam o desempenho (não sendo portanto espectável que otimizações que tenham por alvo o código-máquina se traduzam em melhorias sensíveis).

Uma surpresa (mesmo quando, passada a surpresa inicial, repensamos a questão) é a constatação de que houve uma melhoria de 30% e a taxa de ocupação dos multiprocessadores diminuiu. Esta constatação mostra que nestas arquiteturas complexas o "bom senso" é ilusório.

Na certeza de que as tecnologias surgem e desenvolvem-se para um determinado perfil de utilização, é importante que as implementações realizadas se encaixem nesse perfil para assim obter o melhor rendimento desses recursos e o melhor desempenho de execução. Como tal, a implementação realizada desvia-se desse tipo de perfil, não só pela natureza do problema a tratar, milhões de partículas com múltiplas interações a tratar em cada partícula, mas também porque se conclui que as estruturas de dados usadas não se adequam à arquitetura e aos algoritmos, pois resultam em padrões de acesso que não exibem localidades de referência (pelo menos, espacial).

5.2 Trabalho futuro

Conhecidas as tecnologias GPUs disponíveis no mercado, não existe atualmente dispositivos que por si só tenham capacidades suficientes e respondam em tempo útil às simulações desejadas e como tal, faz todo o sentido realizar este tipo de simulações em aglomerados de recurso, ou seja, através da utilização de *clusters* de recursos computacionais. Assim sendo, seria desejável desenvolver uma implementação que permitisse utilizar múltiplos recursos computacionais que no conjunto das partes, tivesse efetivamente todas as capacidades em pleno (memória, largura de banda, baixa latência e capacidade de processamento) para realizar essas simulações. Tais tecnologias já existem ou começam a emergir como por exemplo a NVIDIA ® Maximus [66] e a NVIDIA ® GRID VCA [67].

Bibliografia

[1] W. Stallings, Computer Organization and Architecture Design for Performance 8th Edition, Pearson Education Inc., 2011.

[2] K. Krabbenhoft e M. Vicenta da Silva, *A Formulation for Large-Scale Granular Contact Dynamics Computations using GP-GPUs (Artigo em processo de revisão),* 2014.

[3] K. Krabbenhoft, A. Lyamin, J. Huang e M. Vicente da Silva, "Granular contact dynamics using mathematical programming methods," *ELSEVIER, Computers and Geotechnics,* n.º 43, pp. 165-176, 2012.

[4] J. Huang, M. Vicente da Silva e K. Krabbenhoft, "Three-dimensional granular contact dynamics with rolling resistance," 21 Junho 2012.

[5] K. Krabbenhoft, J. Huang, M. Vicente da Silva e L. A.V., "Granular contact dynamics with particle elasticity," pp. 607-619, 2012.

[6] P. W. Cleary e S. M. L., "Three -Dimensional Modelling Of Industrial Granular Flows," em *2º International Conference on CFD in the Minerals and Process Industries*, Australia, 1999.

[7] M. L. Sawley e P. W. Cleary, "Une méthode d'éléments discrets parallélisés pour les simulations d'écoulements granulaires industrielles," em *CSIRO Mathematical & Information Scienses, ,* Clayton, Australia, 1999.

[8] J. A. Ferrez e T. M. Liebling, "Parallel DEM Simulations of Granular Materials," 2001.

[9] M. Renouf, F. Dubois e P. Alart, "A parallel version of the non smooth contact dynamics algorithm applied to the simulation of granular media," *Journal of Computational and Applied Mathematics 168,* pp. 375-382, 2004.

[10] A. Patraa, A. Bauera, C. Nichitab, E. Pitmanb, M. Sheridanc, M. Bursikc, B. Ruppc, A. Webberc, A. Stintonc, L. Namikawad e C. Renschlerd, "Parallel adaptive numerical simulation of dry avalanches over natural terrain," *Journal of Volcanology and Geothermal Research 139,* pp. 1-21, 2005.

[11] A. Maknickasa, A. Kačeniauskasa, R. Balevičiusb, R. Kačianauskasb e A. Džiugysc, "Parallel implementation of DEM for visco-elastic granular media," em *CMM-2005 – Computer Methods in Mechanics*, Czestochowa, Poland, 2005.

[12] A. Maknickasa, A. Kačeniauskasa, R. Balevičiusb, R. Kačianauskasb e A. Džiugysc, "Parallel DEM Software for Simulation of Granular Media," *INFORMATICA*, vol. 17, n.º 12, pp. 207-224, 2006.

[13] J. H. Walther e I. F. Sbalzarini, "Large-scale parallel discrete element simulations of granular flow," *International Journal for Computer-Aided Engineering and Software*, vol. 26, n.º 6, pp. 688-697, 2009.

[14] D. Negrut, A. Tasora, M. Anitescu, H. Mazhar, T. Heyn e A. Pazouki, "Solving Large Multibody Dynamics Problems on the GPU," em *GPU Computing Gems JAde Edition*, Morgan Kaufmann, 2011.

[15] D. Negrut, A. Tasora, M. Anitescu, H. Mazhar, T. Heyn e A. Pazouki, "Simulation of Multibody Dynamics Leveraging New Numerical Methods and Multiprocessor Capabilities," em *Proceedings of 2011 NSF Engineering Research and Innovation*, Atlanta, Georgia, 2011.

[16] R. Yasuda, T. Harada e Y. Kawaguchi, "Real-time Simulation of Granular Materials Using Graphics Hardware," em *Fifth International Conference on Computer Graphics, Imaging and Visualization*, 2008.

[17] S. Boyd, N. Parikh, E. Chu e Borja, "Distributed Optimization and Statistical Learning via the Alternating Direction Method of Multipliers," *Foundations and Trends in Machine Learning*, vol. 3, pp. 1-122, 2010.

[18] J. Nocedal e S. Wright, Numerical Optimization, 2nd ed., New York: Springer, 2006.

[19] I. Griva, S. G. Nash e A. Sofer, Linear and Nonlinear Optimization, 2nd. ed., SIAM, 2009.

[20] Intel Corporation, "Moore's Law Inspires Intel Innovation," 2013. [Online]. Available: http://www.intel.com/content/www/us/en/silicon-innovations/moores-law-technology.html. [Acedido em 27 Maio 2013].

[21] Prometeus GmbH , "TOP 500 Supercomputers Site - November 2012," 2013. [Online]. [Acedido em 27 Maio 2013].

[22] L. Null e J. Lobur, The Essential of Computer Organization and Architecture, 2ºEd, Jones and Bartlett Publishers, Inc, 2006.

[23] M. Flynn, "Very High Speed Computing Systems," *Proceedings IEEE*, 1966.

[24] Chhnang, "Wood Come Chhnang blog," 2011. [Online]. Available: http://blog.csdn.net/muxiqingyang/article/details/6723496. [Acedido em 28 Maio 2013].

[25] OpenMP, "OpenMP Application Program Interface V4.0," Março 2013. [Online]. Available: http://www.openmp.org/mp-documents/OpenMP_4.0_RC2.pdf. [Acedido em 29 Maio 2013].

[26] L. Palma, "Intel® Threading Building Blocks, OpenMP ou threads nativas?," 2013. [Online]. Available: http://software.intel.com/pt-br/articles/intel-threading-building-blocks-openmp-or-native-threads. [Acedido em 29 Maio 2013].

[27] J. B. Polkinghorne e M. A. Desnoyers, "Application Specific Integrated Circuit". EUA Patente 4816823, 1989.

[28] National Intruments, "Understanding Parallel Hardware: Multiprocessors, Hyperthreading, Dual-Core,

Multicore and FPGAs," 06 Dezembro 2011. [Online]. Available: http://www.ni.com/white-paper/6097/en#toc4. [Acedido em 27 Maio 2013].

[29] National Instruments, "Optimizing your LabVIEW FPGA VIs: Parallel Execution and Pipelining," 03 Março 2012. [Online]. Available: http://www.ni.com/white-paper/3749/en. [Acedido em 27 Maio 2013].

[30] T. Chen, R. Raghavan, J. Dale e E. Iwata, "Cell Broadband Engine Architecture and its first implementation, A performance view," 29 Novembro 2005. [Online]. Available: http://www.ibm.com/developerworks/power/library/pa-cellperf/. [Acedido em 27 Maio 2013].

[31] D. B. Kirk e W.-m. W. Hwu, Programming Massively Parallel Processors, A Hands-on Approach, 2ª Ed ed., Morgan Kaufman, 2013.

[32] NVIDIA, "GeForce 256, The World's First GPU," 2013. [Online]. Available: http://www.nvidia.com/page/geforce256.html. [Acedido em 05 Junho 2013].

[33] NVIDIA, "CUDA C Programming Guide," NVIDIA, 2012a.

[34] NVIDIA, "NVIDIA GeForce 8800 GPU Architecture Overview," Novembro 2006. [Online]. Available: http://www.nvidia.com/page/8800_tech_briefs.html. [Acedido em 05 Junho 2013].

[35] NVIDIA, "PRODUCT SITEMAP," 2013. [Online]. Available: http://www.nvidia.com/page/products.html. [Acedido em 06 Junho 2013].

[36] AMD, "Products," 2013. [Online]. Available: http://www.amd.com/br/products/Pages/Products.aspx. [Acedido em 05 Junho 2013].

[37] NVIDIA, "Whitepaper NVIDIA's Next Generation CUDATM Compute Architecture: Fermi," 2009. [Online]. Available: http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf. [Acedido em 11 junho 2013].

[38] NVIDIA, "Whitepaper NVIDIA's Next Generation CUDA Compute Architecture: Kepler GK110," 2012. [Online]. Available: http://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf. [Acedido em 12 Junho 2013].

[39] N. Inc, "NVIDIA Developper Zone," 21 02 2014. [Online]. Available: http://devblogs.nvidia.com/parallelforall/5-things-you-should-know-about-new-maxwell-gpu-architecture/. [Acedido em 01 09 2014].

[40] NVIDIA Corporation, "Whitepaper NVIDIA GeForce GTX 750 Ti," 01 09 2014. [Online]. Available: http://international.download.nvidia.com/geforce-com/international/pdfs/GeForce-GTX-750-Ti-Whitepaper.pdf. [Acedido em 01 09 2014].

[41] M. Chiappetta, "ATI Radeon HD 4850 and 4870: RV770 Has Arrived," 2008. [Online]. Available: http://hothardware.com/Reviews/ATI-Radeon-HD-4850-and-4870-RV770-Has-Arrived/?page=2. [Acedido em 10 Junho 2013].

[42] S. Wasson, "AMD's Radeon HD 5870 graphics processor," 2009. [Online]. Available: http://techreport.com/review/17618/amd-radeon-hd-5870-graphics-processor/5. [Acedido em 10 Junho 2013].

[43] S. Ryan, “AMD's Radeon HD 6970 & Radeon HD 6950: Paving The Future For AM,” 2010. [Online]. Available: http://www.anandtech.com/show/4061/amds-radeon-hd-6970-radeon-hd-6950/4. [Acedido em 10 Junho 2013].

[44] S. Lepilov, “AMD Radeon HD 7790 vs. Nvidia GeForce GTX 650 Ti BOOST: Performance Review.,” 2013. [Online]. Available: http://www.xbitlabs.com/articles/graphics/display/radeon-hd-7790-geforce-gtx-650ti-boost_8.html#sect1. [Acedido em 14 junho 2013].

[45] S. Walton, “The Best Graphics Cards: Nvidia vs. AMD Current-Gen Comparison,” 2012. [Online]. Available: http://www.techspot.com/review/603-best-graphics-cards/page12.html. [Acedido em 14 junho 2013].

[46] S. Lepilov, “Even More Speed, Even Lower Price: AMD Radeon HD 7970 GHz Edition 3 GB Graphics Card Review.,” 2012. [Online]. Available: http://www.xbitlabs.com/articles/graphics/display/radeon-hd-7970-ghz-edition_9.html#sect1.

[47] B. M. Muenchen, “GPGPU: comparação de aceleradores AMD, NVIDIA e INTEL utilizando a biblioteca OPENCL,” UNIJUÍ, Brasil, 2013.

[48] T. Shimobaba, T. Ito, N. Masuda, Y. Ichihashi e N. Takada, “Fast calculation of computer-generated-hologram on AMD HD5000 series GPU and OpenCL,” 3 fevereiro 2010. [Online]. Available: http://arxiv.org/abs/1002.0916v2. [Acedido em 14 junho 2013].

[49] K. Komatsu1, K. Sato, Y. Arai, K. Koyama, H. Takizawa e H. Kobayashi, “Evaluating Performance and Portability of OpenCL Programs,” 2010. [Online]. Available: http://vecpar.fe.up.pt/2010/workshops-iWAPT/Komatsu-Sato-Arai-Koyama-Takizawa-Kobayashi.pdf. [Acedido em 2014 junho 2013].

[50] NVIDIA, CUDA C BEST PRACTICES GUIDE, US: NVIDIA, 2012.

[51] Khronos Group, “The OpenCL Specification,” 2012. [Online]. Available: http://www.khronos.org/registry/cl/specs/opencl-1.2.pdf. [Acedido em 18 junho 2012].

[52] NVIDIA, “Developer Zone: OpenCL,” 2013. [Online]. Available: https://developer.nvidia.com/opencl. [Acedido em 18 junho 2013].

[53] AMD, “Developer Central: OpenCL Zone,” AMD, 2013. [Online]. Available: http://developer.amd.com/resources/heterogeneous-computing/opencl-zone/. [Acedido em 21 junho 2013].

[54] M. Shevtsov, “OpenCL*: the advantages of heterogeneous approach,” Intel, 2013. [Online]. Available: http://software.intel.com/en-us/articles/opencl-the-advantages-of-heterogeneous-approach. [Acedido em 21 junho 2013].

[55] OpenACC, “OpenACC Programming Interface,” 17 junho 2013. [Online]. Available: http://www.openacc-standard.org/node/297. [Acedido em 23 junho 2013].

[56] G. M. Amdahl, “Validity of the single processor approach to achieving large scale computing capabilities,” em *AFIPS spring joint computer conference*, IBM Sunnyvale California, 1967.

[57] J. L. Gustafson, “Reavaluating Amdalh's Law,” *Communications of the ACM*, vol. 31, n.º 5, pp. 532-533, May 1988.

[58] K. Karimi, N. G. Dickson e F. Hamze, “A performance Comparison of CUDA and OpenCL,” Cornell

University, 2011.

[59] J. Fang, A. L. Varbanescu e H. Sips, “A Comprehensive Performance Comparison of CUDA and OpenCL,” Delft University of Techonology, Delft, the Netherlands, 2011.

[60] R. Reyes, F. J. J. López Ivan e F. Sande, “Directive-based Programming for GPUs: A Comparative Study,” em *IEEE 14th International Conferences on High Performance Computing and Communications*, 2012.

[61] A. Hart, R. Ansaloni e A. Gray, “Porting and scaling OpenACC applications on massively-parallel, GPU-accelerated supercomputers,” *The European Physical Journal,* 2012.

[62] R. Farber, CUDA Application Design and Development, Morgan Kaufmann.

[63] NVIDIA, “GPU-Accelerated Libraries,” NVIDIA, 2013. [Online]. Available: https://developer.nvidia.com/gpu-accelerated-libraries. [Acedido em 15 junho 2013].

[64] NVIDIA, “Developer Zone, Cuda Toolkit Documentation, Profiler Users Guide,” NVIDIA, 2013. [Online]. Available: http://docs.nvidia.com/cuda/profiler-users-guide/index.html. [Acedido em junho 2013].

[65] U. Ayachit, A. Bauer e A. Chaudhary, ParaView Manual: A Parallel Visualization Application v4.0, ?: ?, 2013.

[66] NVIDIA Inc, “NVIDIA: Multi-GPU Technology,” 09 2014. [Online]. Available: http://www.nvidia.com/object/multi-gpu-technology.html. [Acedido em 22 Setembro 2014].

[67] N. Inc, “NVIDIA: NVIDIA GRID VCA,” Setembro 2014. [Online]. Available: http://www.nvidia.com/object/grid-vca.html. [Acedido em 22 09 2014].

[68] AMD, “AMD History,” 2013. [Online]. Available: http://www.amd.com/us/aboutamd/corporate-information/Pages/timeline.aspx. [Acedido em 05 Junho 2013].

[69] S. Wasson, “AMD's Radeon HD 2900 XT graphics processor, R600 Revealed,” 2007. [Online]. Available: http://techreport.com/review/12458/amd-radeon-hd-2900-xt-graphics-processor. [Acedido em 05 Junho 2013].

[70] E. Chester, “RV770: AMD ATI Radeon HD 4870 - RV770: The Architecture,” 2011. [Online]. Available: http://www.trustedreviews.com/RV770-ATI-Radeon-HD-4870_PC-Component_review_rv770-the-architecture_Page-2. [Acedido em 10 Junho 2013].

[71] A. L. Shimpi e D. Wilson, “The Radeon HD 4850 & 4870,” 2008. [Online]. Available: http://www.anandtech.com/show/2556/3. [Acedido em 10 Junho 2013].

[72] S. Ryan, “AMD Radeon HD 6970 and 6950 Review - The Cayman Architecture Revealed,” 2010. [Online]. Available: http://www.pcper.com/reviews/Graphics-Cards/AMD-Radeon-HD-6970-and-6950-Review-Cayman-Architecture-Revealed?aid=1051. [Acedido em 10 Junho 2013].

[73] AMD, “AMD GRAPHICS CORES NEXT (GCN) ARCHITECTURE,” 2012. [Online]. Available: http://www.amd.com/us/Documents/GCN_Architecture_whitepaper.pdf. [Acedido em 10 Junho 2013].

[74] T. Sandhu, “Review: AMD Radeon HD 7970 3GB,” 22 Novembro 2011. [Online]. Available: http://hexus.net/tech/reviews/graphics/33031-amd-radeon-hd-7970-3gb/?page=2. [Acedido em 10 Junho 2013].

[75] D. Kanter, “NVIDIA’s GT200: Inside a Parallel Processor,” 2008. [Online]. Available: http://www.realworldtech.com/gt200/7/. [Acedido em 11 junho 2013].

[76] J. H. Walther e I. F. Sbalzarini, “Large-scale Parallel Discrete Element Simulations of Granular Flow,” 2007.

[77] G. Akinci, M. Ihmsen, N. Akinci e M. Teschner, “Parallel Surface Reconstruction for Particle-Based Fluids,” *COMPUTER GRAPHICS forum*, vol. 0, n.º 0, pp. 1-12, 2012.

Anexo A: Profiling detalhado dos kernels

O kernel Update

Conforme se pode observar pela Figura A.1 este *kernel* está a ser executado numa configuração de 3 blocos de 512 *threads* por cada multiprocessador, utilizando 26 registos por cada *thread* e consome 2kiB de memória partilhada por bloco. Possui baixa eficiência nas operações de *load* e *store* sobre a memória global obtendo melhores rendimentos de eficiência das operações sobre a memória partilhada e execução dos *warps*. A maior parte do tempo consumido pelo *kernel* é referente a acessos a memória sendo as restantes operações residuais relativamente ao tempo total.

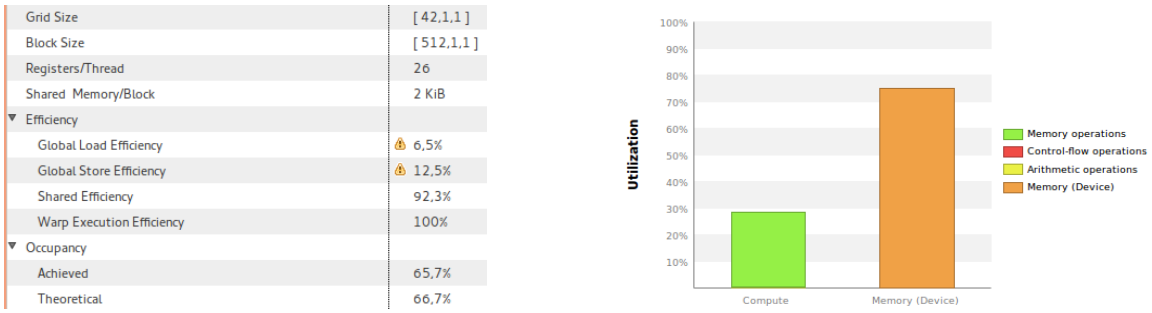


Figura A.1: *Profiling* do *kernel* Update

A taxa de ocupação obtida de 65%, está a ser limitada pelo número de registos utilizados (Figura A.2), 13312 registos por bloco, levando que os multiprocessadores apenas consigam executar 2 blocos simultaneamente num limite possível de 8 blocos ou seja, executar 32 *warps* de um limite possível de 48 *warps*.

.

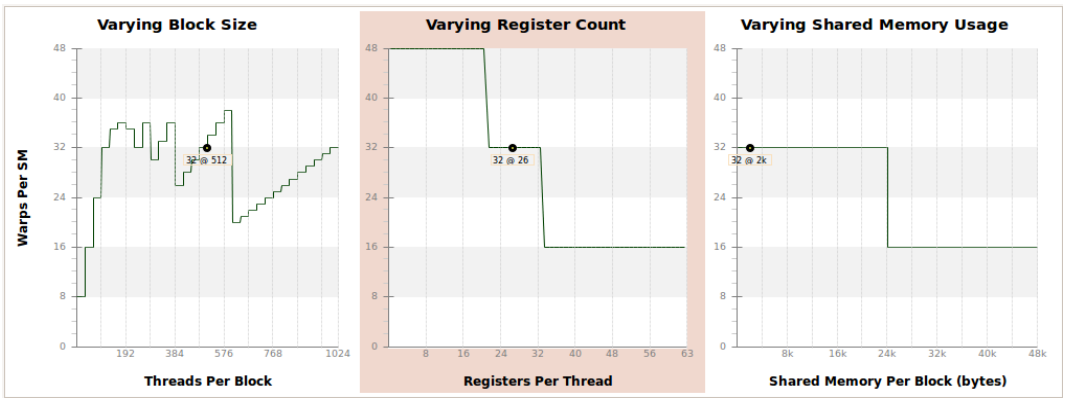


Figura A.2: Variação da taxa de ocupação relativamente ao número de blocos, registos e memória partilhada.

Uma possível solução para contornar esta limitação na taxa de ocupação passa por reduzir o número de registos utilizados contudo não é garantido que se obtenha melhores desempenhos pois nem sempre um aumento da taxa de ocupação traz benefícios de desempenho.

Observada a Figura A.3 pode-se verificar que a latência de execução das instruções do *kernel* resulta da não disponibilidade dos dados de *output* de outras instruções o que pode reduzir o desempenho obtido pelo *kernel*. Aumentando o paralelismo ao nível das instruções poder-se-á reduzir esta dependência e melhorar o desempenho.

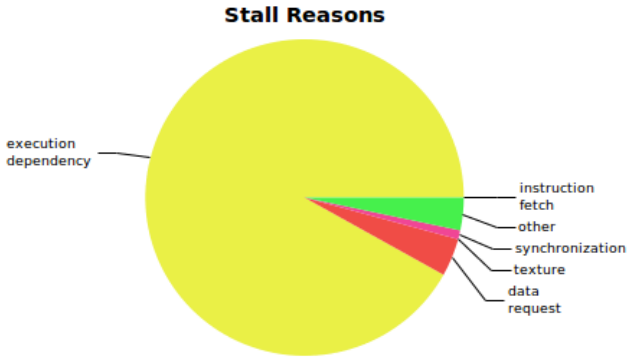


Figura A.3: Origens das latências obtidas do kernel Update

Para uma melhor compreensão do funcionamento do *kernel*, deve-se efetuar uma análise ao tipo de instruções que são executadas. Conforme se pode verificar na Figura A.4 as instruções mais utilizadas são as referentes a operações de *load*/*store* sobre a memória, seguindo-se as operações aritméticas e controlo de fluxo. Dentro das instruções aritméticas as mais representativas são operações de vírgula flutuante nomeadamente operações de adição e multiplicação.

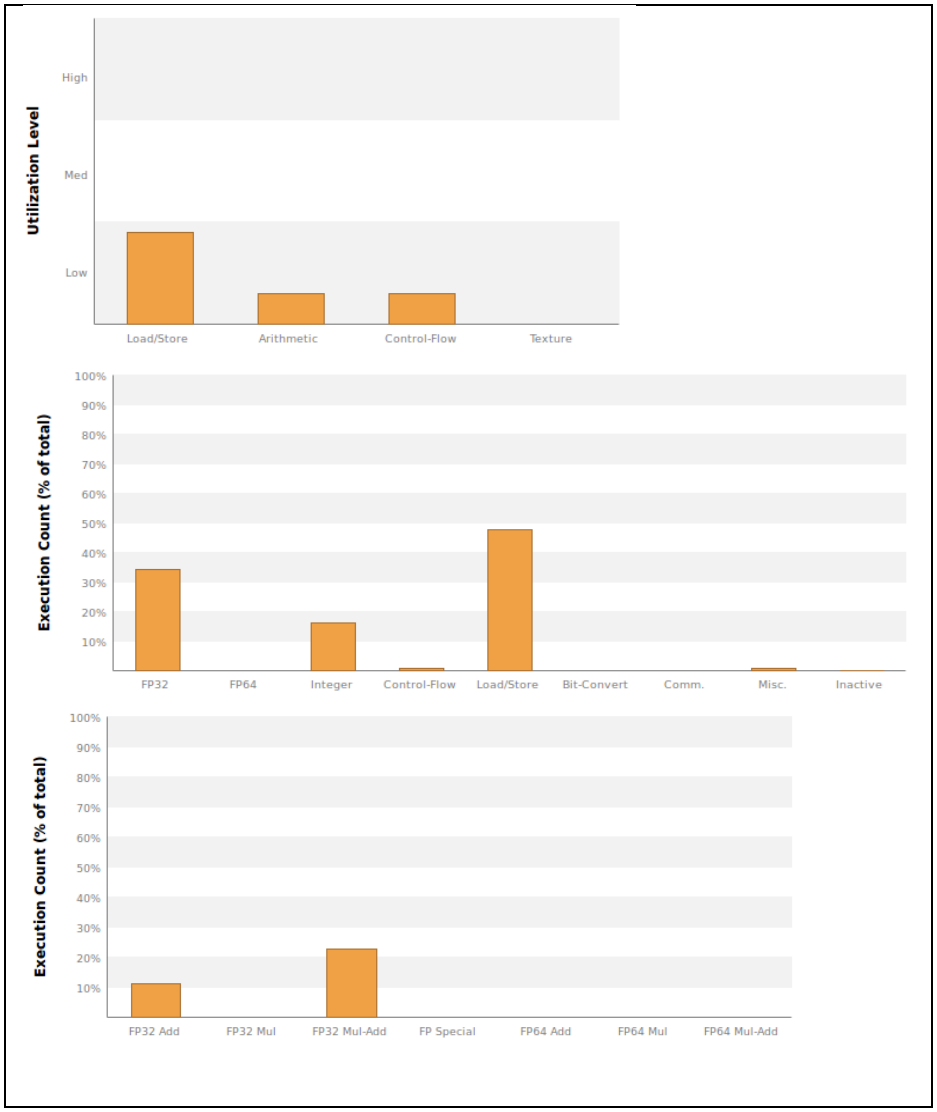
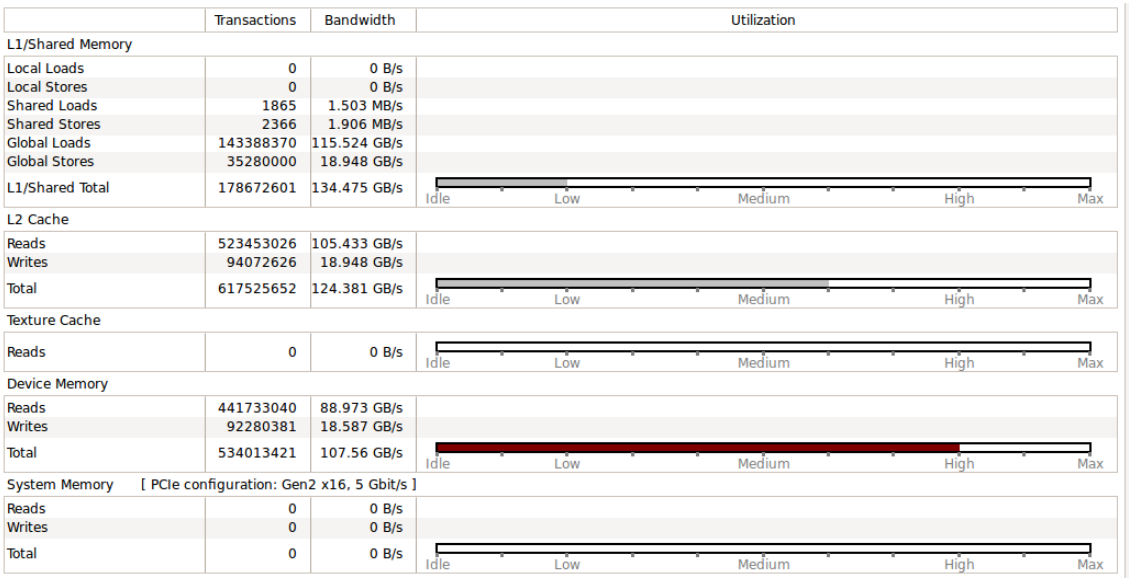


Figura A.4:Tipo de operações utilizadas pelo *kernel Update*

Relativamente à utilização da memória pode-se observar pelos resultados obtidos na Figura A..5 que o maior número de transações decorre sobre a memória global e na utilização da *cache* L2. Possíveis otimizações passam por guardar os dados na memória partilhada ao invés da memória global e explorar se aumentando a dimensão da *cache* na memória L1 se obtém melhores desempenhos.



	Transactions	Bandwidth	Utilization
L1/Shared Memory			
Local Loads	0	0 B/s	
Local Stores	0	0 B/s	
Shared Loads	1865	1.503 MB/s	
Shared Stores	2366	1.906 MB/s	
Global Loads	143388370	115.524 GB/s	
Global Stores	35280000	18.948 GB/s	
L1/Shared Total	178672601	134.475 GB/s	Idle Low Medium High Max
L2 Cache			
Reads	523453026	105.433 GB/s	
Writes	94072626	18.948 GB/s	
Total	617525652	124.381 GB/s	Idle Low Medium High Max
Texture Cache			
Reads	0	0 B/s	Idle Low Medium High Max
Device Memory			
Reads	441733040	88.973 GB/s	
Writes	92280381	18.587 GB/s	
Total	534013421	107.56 GB/s	Idle Low Medium High Max
System Memory [PCIe configuration: Gen2 x16, 5 Gbit/s]			
Reads	0	0 B/s	
Writes	0	0 B/s	
Total	0	0 B/s	Idle Low Medium High Max

Figura A..5: Largura de banda alcançada nos acessos a memória

O kernel GlobalSol

Conforme se pode observar pela Figura A..6 este *kernel* está a ser executado numa configuração de 3 blocos de 448 *threads* por cada multiprocessador, utilizando 23 registos por cada *thread* e consome 1,7KiB de memória partilhada por bloco. Possui baixa eficiência nas operações de *load* e *store* sobre a memória global mas alcança boa eficiência das operações sobre a memória partilhada. Relativamente à eficiência de execução dos *warps* alcança os 73,3% o que pode revelar algum grau de divergência na execução dos *warps*. A maior parte do tempo consumido pelo *kernel* é referente a acessos a memória sendo as restantes operações residuais relativamente ao tempo total (Figura A..6).

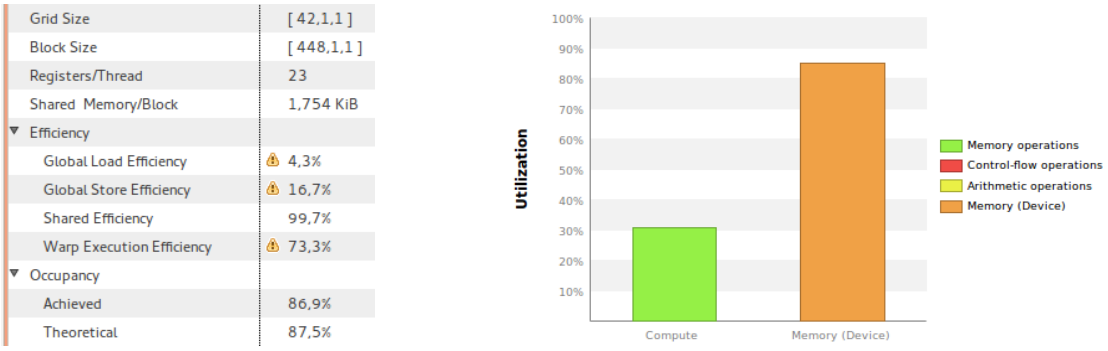


Figura A..6: *Profiling* geral dos *kernel GlobalSol*

Revela uma taxa de ocupação de 87,5% estando a ser limitada pelo número de *threads* por bloco e pelo número de registos por *thread*. Fica limitado a 42 *warps* por multiprocessador ao invés dos 48 possíveis. Na Figura A..7 pode-se encontrar o impacto da variação do número de registos e do número de *threads* e a sua implicação na taxa de

utilização. A solução para aumentar a taxa de ocupação passa por reduzir o número de registos utilizados bem como aumentar o número de *threads* por bloco. Deve-se de igual forma procurar aumentar a eficiência de execução dos *warps* sendo necessário para tal minimizar ramos divergentes de execução.

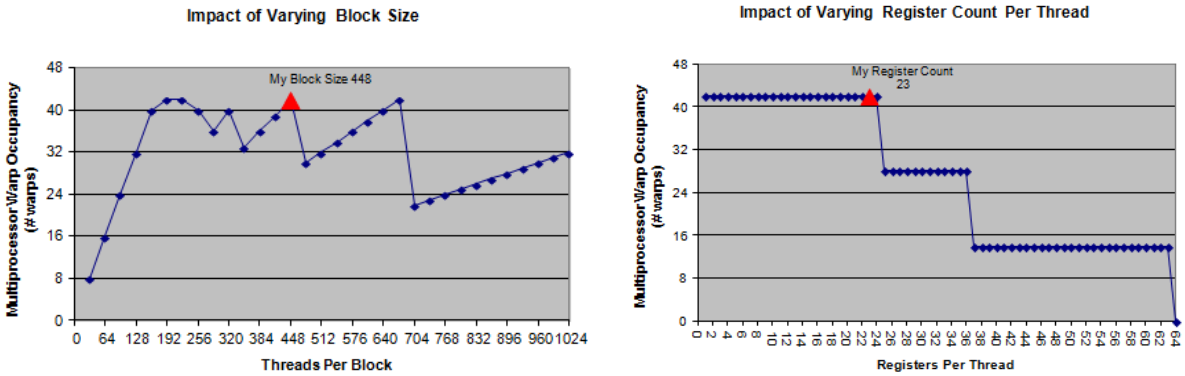
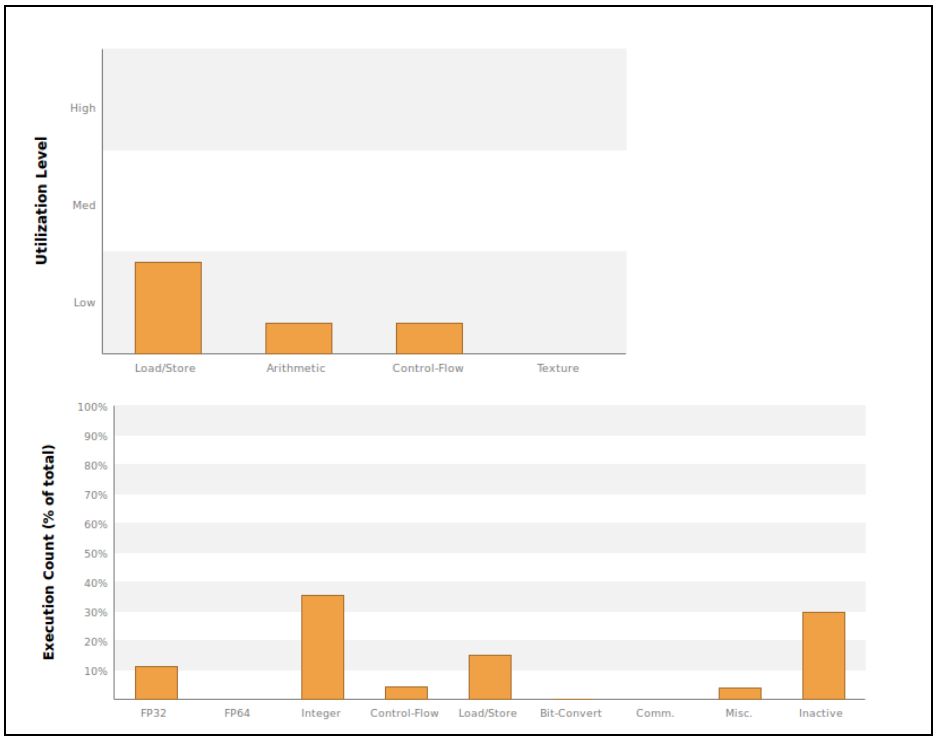


Figura A..7: Taxa de ocupação obtida variando o tamanho do bloco e do número de registos.

Relativamente ao tipo de instruções utilizadas, pode-se observar pela Figura A.8 que existe uma maior utilização de instruções sobre operações de *load/store*. Deve-se realçar o número de instruções não executadas devido ao facto das *threads* estarem inativos consequência da divergência da execução.



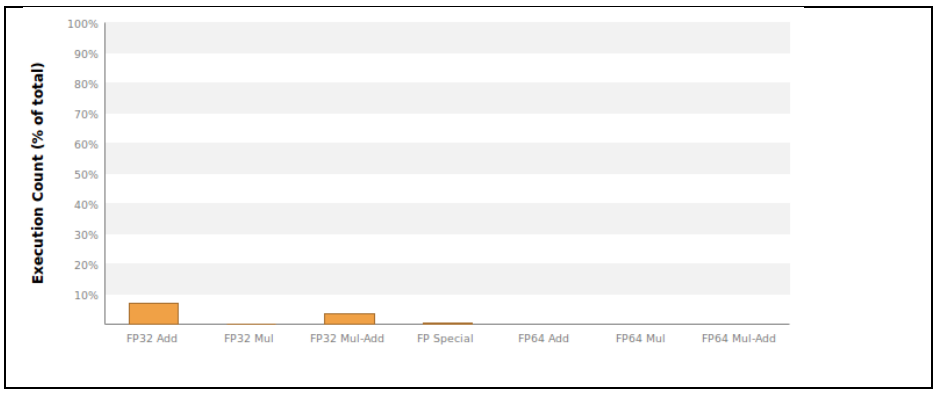


Figura A.8: Tipo de instruções utilizados no *kernel*

Relativamente à utilização da memória pode-se observar pelos resultados obtidos na Figura A.9 que o maior número de transações decorre sobre a memória global e *cache* L2. Possíveis otimizações passam por guardar os dados na memória partilhada ao invés da memória global e explorar se aumentando a dimensão da *cache* na memória L1 se obtém melhores desempenhos.

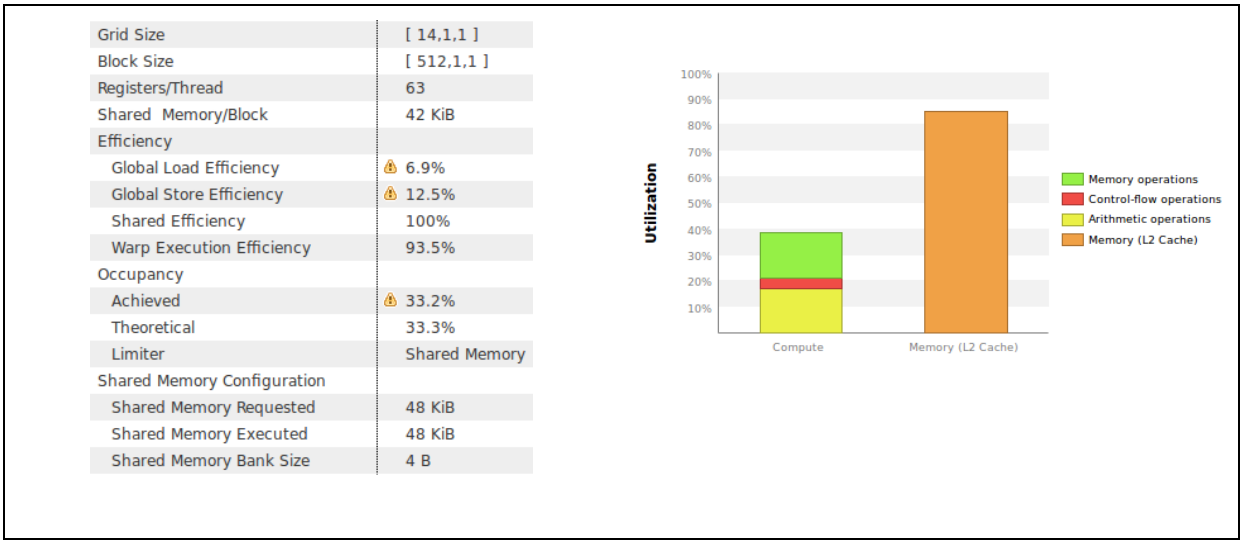
	Transactions	Bandwidth	Utilization
L1/Shared Memory			
Local Loads	0	0 B/s	
Local Stores	0	0 B/s	
Shared Loads	68208	72.918 MB/s	
Shared Stores	36540	39.063 MB/s	
Global Loads	146342182	156.448 GB/s	
Global Stores	1227744	1.309 GB/s	
L1/Shared Total	147674674	157.869 GB/s	Idle Low Medium High Max
L2 Cache			
Reads	514479854	137.502 GB/s	
Writes	4904430	1.311 GB/s	
Total	519384284	138.813 GB/s	Idle Low Medium High Max
Texture Cache			
Reads	0	0 B/s	Idle Low Medium High Max
Device Memory			
Reads	470858661	125.844 GB/s	
Writes	4903988	1.311 GB/s	
Total	475762649	127.155 GB/s	Idle Low Medium High Max
System Memory [PCIe configuration: Gen2 x16, 5 Gbit/s]			
Reads	0	0 B/s	
Writes	0	0 B/s	
Total	0	0 B/s	Idle Low Medium High Max

Figura A.9: Largura de banda e taxa de utilização dos acessos a memória.

O kernel CalcA

Conforme se pode observar pela Figura A.10 este *kernel* está a ser executado numa configuração de 1 bloco de 512 *threads* por cada multiprocessador, utilizando 63 registos por cada *thread* e consome 42 KiB de memória partilhada por bloco. Possui baixa eficiência nas operações de *load* e *store* sobre a memória global mas alcança boa eficiência das operações sobre a memória partilhada. Relativamente à eficiência de execução dos *warps* alcança os 93,5% o que revela baixo grau de divergência na execução dos *warps*. A maior parte do tempo consumido pelo *kernel* é referente a acessos à *cache* L2, contudo, relativamente aos *kernels* já referidos apresenta percentagem superior no que se refere a operações aritméticas e

controlo do fluxo das operações. (Figura A.10, lado direito).



Grid Size	[14,1,1]
Block Size	[512,1,1]
Registers/Thread	63
Shared Memory/Block	42 KiB
Efficiency	
Global Load Efficiency	⚠ 6.9%
Global Store Efficiency	⚠ 12.5%
Shared Efficiency	100%
Warp Execution Efficiency	93.5%
Occupancy	
Achieved	⚠ 33.2%
Theoretical	33.3%
Limiter	Shared Memory
Shared Memory Configuration	
Shared Memory Requested	48 KiB
Shared Memory Executed	48 KiB
Shared Memory Bank Size	4 B

Figura A.10: *Profiling* geral do *kernel* CalcA

Revela uma fraca taxa de ocupação, abaixo dos 34%, estando a ser limitada pelo número de *threads* por bloco e pelo número de registos por *thread*. Fica limitado a 16 *warps* por multiprocessador ao invés dos 48 possíveis. Na Figura A.11 pode-se encontrar o impacto da variação do número de registos e do número de *threads* e a sua implicação na taxa de utilização. Possíveis soluções para aumentar a taxa de ocupação passam por reduzir o número de registos utilizados e memória partilhada utilizada possivelmente refazendo o código do *kernel*.

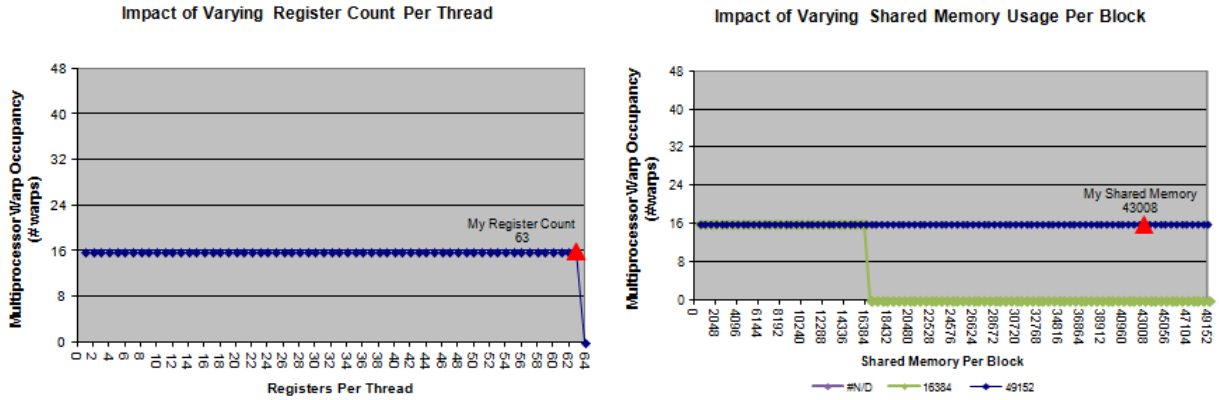


Figura A.11: Variação da taxa de ocupação relativamente ao número de registos e memória partilhada

Ainda relativamente à latência de execução das instruções pode-se observar que estas devem-se em grande percentagem à espera de *outputs* produzidos por outras instruções (Figura A.12) sendo uma possível solução aumentar o paralelismo de execução de s instruções.

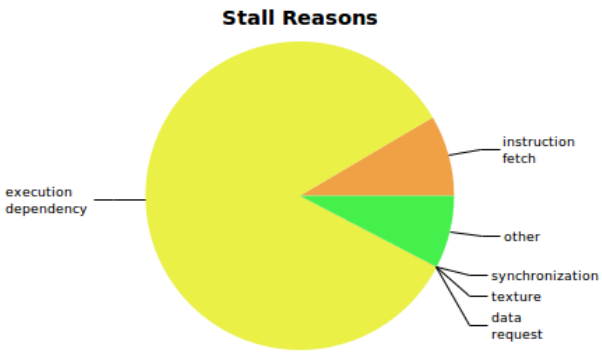


Figura A.12: Origens da latência de execução das instruções.

Relativamente ao tipo de instruções utilizadas, pode-se observar pela Figura A.13 que existe algum equilíbrio entre as operações de acesso a memória e aritméticas observando ainda existência de operações de controlo de fluxo.

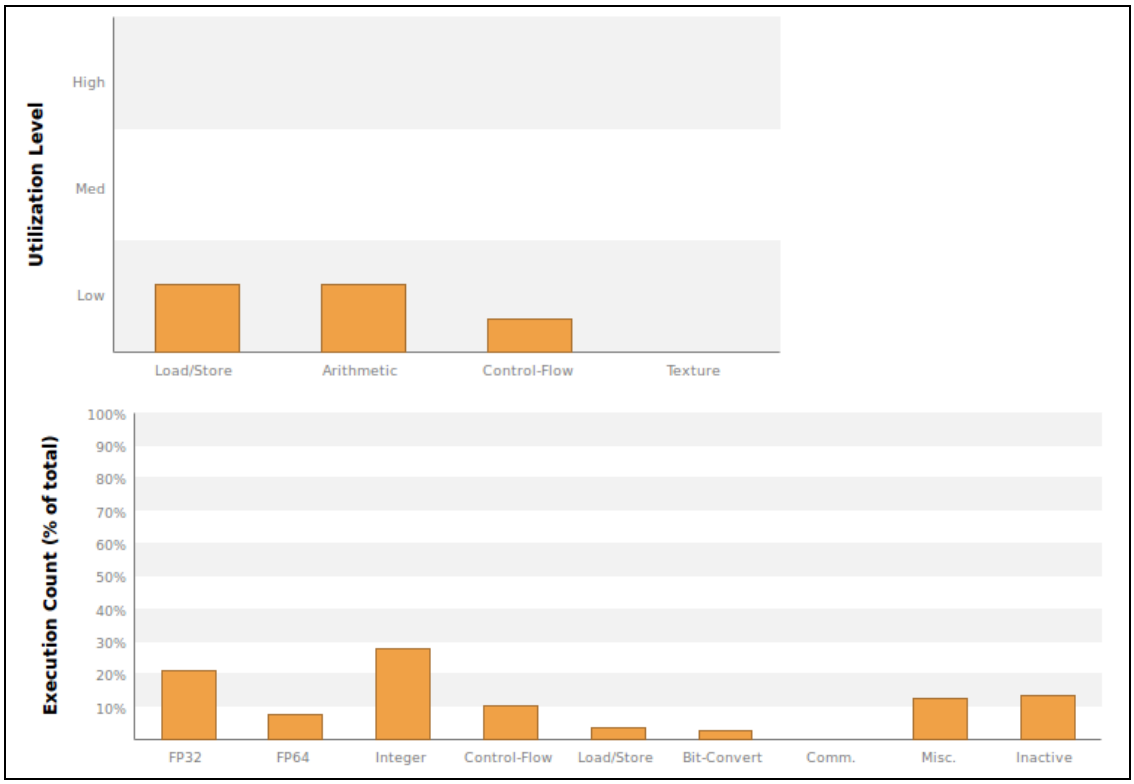


Figura A.13: Tipo de instruções utilizadas

Relativamente à utilização da memória pode-se observar pelos resultados obtidos na Figura A.14 que o maior número de transações decorre sobre a *cache* L2. Possíveis otimizações passam por guardar os dados na memória partilhada ao invés da memória global e explorar se aumentando a dimensão da *cache* na memória L1 se obtém melhores desempenhos.

Memory Bandwidth And Utilization

The following table shows the memory bandwidth used by this kernel for the various types of memory on the device. The table also shows the utilization of each memory type relative to the maximum throughput supported by the memory. More...

	Transactions	Bandwidth	Utilization
L1/Shared Memory			
Local Loads	245055	357.237 MB/s	
Local Stores	245056	357.238 MB/s	
Shared Loads	0	0 B/s	
Shared Stores	4704	6.857 MB/s	
Global Loads	109427055	159.521 GB/s	
Global Stores	35288064	34.284 GB/s	
L1/Shared Total	145209934	194.526 GB/s	Idle Low Medium High Max
L2 Cache			
Reads	419028222	152.713 GB/s	
Writes	94822016	34.557 GB/s	
Total	513850238	187.27 GB/s	Idle Low Medium High Max
Texture Cache			
Reads	0	0 B/s	Idle Low Medium High Max
Device Memory			
Reads	266204340	97.017 GB/s	
Writes	34281673	12.494 GB/s	
Total	300486013	109.511 GB/s	Idle Low Medium High Max
System Memory [PCIe configuration: Gen2 x16, 5 Gbit/s]			
Reads	0	0 B/s	
Writes	0	0 B/s	
Total	0	0 B/s	Idle Low Medium High Max

Figura A.14: Taxa de utilização e largura de banda consumidas no *kernel* CalcA.

O kernel Local2Global

Conforme se pode observar pela Figura A.15 este *kernel* está a ser executado numa configuração de 1 bloco de 512 *threads* por cada multiprocessador, utilizando 63 registos por cada *thread* e consome 42 KiB de memória partilhada por bloco. Possui baixa eficiência nas operações de *load* e *store* sobre a memória global mas alcança máxima eficiência das operações sobre a memória partilhada. Alcança igualmente máxima eficiência de execução dos *warps*. A maior parte do tempo consumido pelo *kernel* é referente a acessos à *cache* L2 (Figura A.15, lado direito).

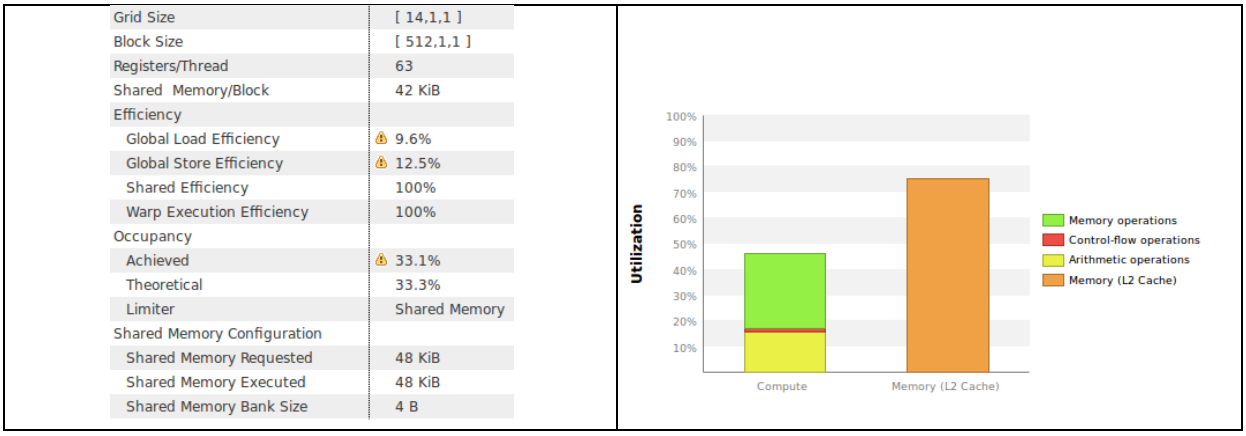


Figura A.15: *Profiling* geral do *kernel* Local2Global

Revela uma fraca taxa de ocupação, cerca de 33%, estando a ser limitada pelo número

de *threads* por bloco e pelo número de registos por *thread*. Fica limitado a 16 *warps* por multiprocessador ao invés dos 48 possíveis. De forma semelhante ao que se passa no *kernel CalcA*, na Figura A.11 pode-se encontrar o impacto da variação do número de registos e do número de *threads* e a sua implicação na taxa de utilização. Possíveis soluções para aumentar a taxa de ocupação passam por reduzir o número de registos e memória partilhada utilizada eventualmente refazendo o código do *kernel*.

Também relativamente à latência de execução das instruções pode-se observar que estas devem-se em grande percentagem à espera de *outputs* produzidos por outras instruções (Figura A.16) sendo uma possível solução aumentar o número de operações ao nível das instruções.

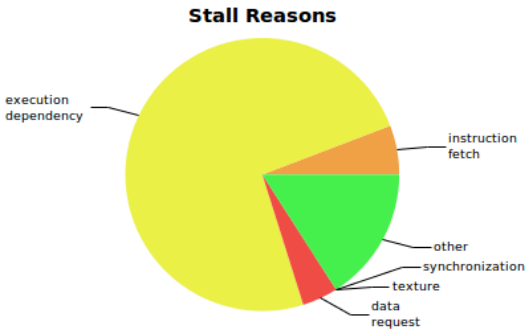


Figura A.16: Latência de execução de instruções do *kernel* Local2Global

Relativamente ao tipo de instruções utilizadas, pode-se observar pela Figura A.17 que existe um ligeiro acréscimo de operações sobre a memória relativamente a instruções aritméticas observando ainda existência de instruções de controlo de fluxo.

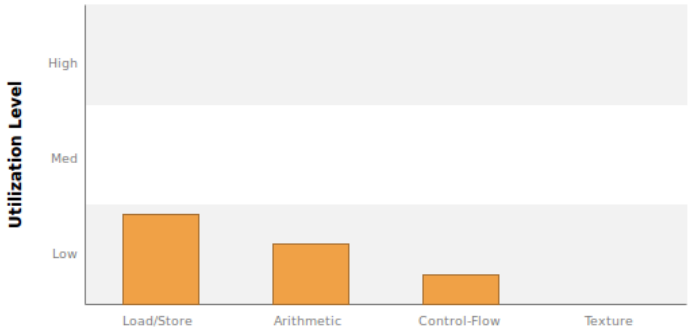


Figura A.17: Tipo de instruções utilizadas no *kernel Local2Global*

Relativamente à utilização da memória pode-se observar pelos resultados obtidos na Figura A.18 que o maior número de transações decorre sobre a *cache* L2 e a memória global, realçando o número de operações de escrita relativamente às operações de leitura. Possíveis otimizações passam por guardar os dados na memória partilhada ao invés da memória global e explorar se aumentando a dimensão da *cache* na memória L1 se obtém melhores desempenhos.

	Transactions	Bandwidth	Utilization
L1/Shared Memory			
Local Loads	6127072	22.273 GB/s	
Local Stores	6126399	22.27 GB/s	
Shared Loads	0	0 B/s	
Shared Stores	4704	17.1 MB/s	
Global Loads	48584861	176.611 GB/s	
Global Stores	35288064	85.491 GB/s	
L1/Shared Total	96131100	306.662 GB/s	Idle Low Medium High Max
L2 Cache			
Reads	81067018	73.672 GB/s	
Writes	107987584	98.137 GB/s	
Total	189054602	171.809 GB/s	Idle Low Medium High Max
Texture Cache			
Reads	0	0 B/s	Idle Low Medium High Max
Device Memory			
Reads	47842130	43.478 GB/s	
Writes	63797415	57.978 GB/s	
Total	111639545	101.455 GB/s	Idle Low Medium High Max
System Memory [PCIe configuration: Gen2 x16, 5 Gbit/s]			
Reads	0	0 B/s	
Writes	0	0 B/s	
Total	0	0 B/s	Idle Low Medium High Max

Figura A.18: Largura de banda e taxa de utilização dos acessos a memória

O kernel LocalMin

Conforme se pode observar pela Figura A.19 este *kernel* está a ser executado numa configuração de 2 blocos de 384 *threads* por cada multiprocessador, utilizando 44 registos por cada *thread* e consome 22,5 KiB de memória partilhada por bloco. Possui baixa eficiência nas operações de *load* e *store* sobre a memória global mas alcança máxima eficiência das operações sobre a memória partilhada. Alcança uma eficiência de execução dos *warps* de 44% o que revela um certo grau de divergência. A maior parte do tempo consumido pelo *kernel* é referente a acessos à memória global, contudo as operações sobre a memória são de menor dimensão comparativamente às operações aritméticas. De realçar o número de operações para o controlo do fluxo de execução (Figura A.19, lado direito).

Grid Size	[28,1,1]
Block Size	[384,1,1]
Registers/Thread	44
Shared Memory/Block	22.5 KiB
Efficiency	
Global Load Efficiency	⚠ 9.4%
Global Store Efficiency	⚠ 12.5%
Shared Efficiency	100.1%
Warp Execution Efficiency	⚠ 44.1%
Occupancy	
Achieved	⚠ 24.6%
Theoretical	25%
Limiter	Registers

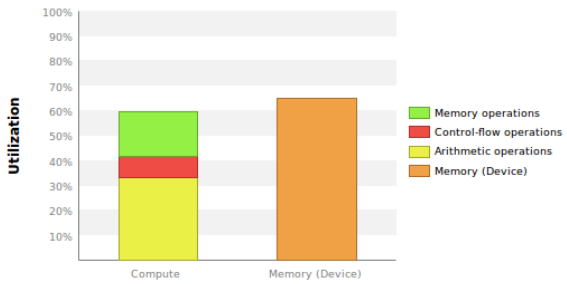


Figura A.19: *Profiling* geral ao *kernel* LocalMin

Revela uma fraca taxa de ocupação, cerca de 25%, estando a ser limitada

principalmente pelo número de registos e pelo número de *threads* por bloco, ficando limitado a 12 *warps* por multiprocessador ao invés dos 48 possíveis. Tal impacto pode ser observado na Figura A.20. Possíveis soluções para aumentar a taxa de ocupação passam por reduzir o número de registos e aumentar o número de *threads* por *kernel*.

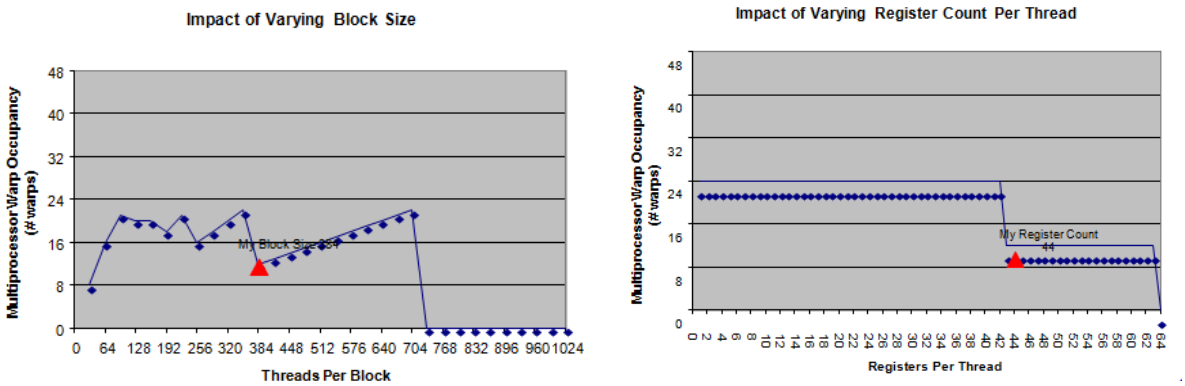


Figura A.20: Variação da taxa de ocupação relativamente ao número de *threads* bloco e número de registos

Relativamente à latência de execução das instruções pode-se observar pela Figura A.21 que existe uma dependência quer do *fetch* de instruções como também do *output* resultante da sua execução.

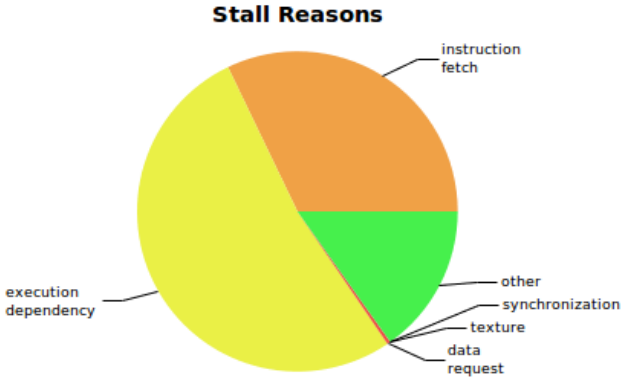


Figura A.21: Latência de execução de instruções

Relativamente ao tipo de instruções utilizadas, pode-se observar pela Figura A.22 que as instruções aritméticas são as mais utilizadas e que existe uma grande percentagem de tempo inativo das *threads* eventualmente devido a fatores de divergência.

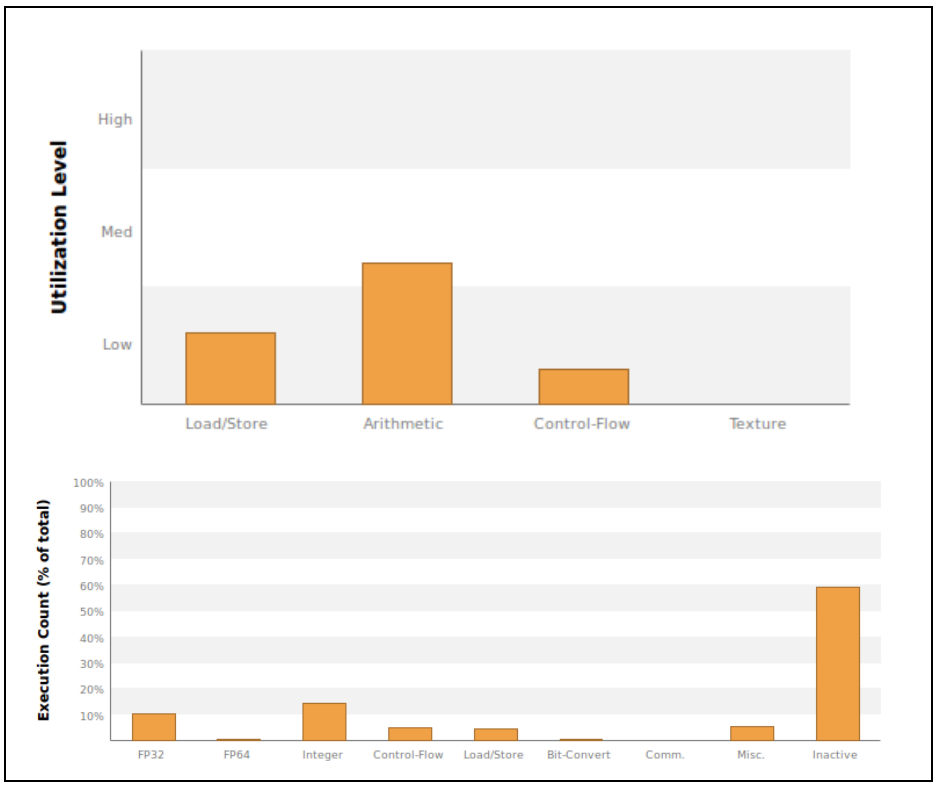


Figura A.22: Tipo de instruções utilizadas pelo *kernel* LocalMin

Relativamente à utilização da memória pode-se observar pelos resultados obtidos na Figura A.23 que o maior número de transações decorre sobre a memória global e a *cache* L2. Possíveis otimizações passam por guardar os dados na memória partilhada ao invés da memória global e explorar se aumentando a dimensão da *cache* na memória L1 se obtém melhores desempenhos.

Memory Bandwidth And Utilization

The following table shows the memory bandwidth used by this kernel for the various types of memory on the device. The table also shows the utilization of each memory type relative to the maximum throughput supported by the memory. More...

	Transactions	Bandwidth	Utilization
L1/Shared Memory			
Local Loads	0	0 B/s	
Local Stores	0	0 B/s	
Shared Loads	1119786	6.463 GB/s	
Shared Stores	5755536	33.22 GB/s	
Global Loads	26034602	150.267 GB/s	
Global Stores	14699160	56.559 GB/s	
L1/Shared Total	47609084	246.509 GB/s	Idle Low Medium High Max
L2 Cache			
Reads	47238640	68.163 GB/s	
Writes	39196910	56.559 GB/s	
Total	86435550	124.722 GB/s	Idle Low Medium High Max
Texture Cache			
Reads	0	0 B/s	Idle Low Medium High Max
Device Memory			
Reads	40360166	58.238 GB/s	
Writes	19958954	28.8 GB/s	
Total	60319120	87.037 GB/s	Idle Low Medium High Max
System Memory [PCIe configuration: Gen2 x16, 5 Gbit/s]			
Reads	0	0 B/s	
Writes	0	0 B/s	
Total	0	0 B/s	Idle Low Medium High Max

Figura A.23: Utilização da memória pelo kernel LocalMin

Anexo B: Implementações otimizadas

Kernel GlobalSol

Implementação 1: sem mecanismos de sincronização e acesso direto à memória global.

```
1.  __global__ void GlobalSol(int nrparticles, int *dev_Cxadj, int *dev_Cadjncy,
2.                            REAL *dev_displacements, REAL *dev_forces,
3.                            REAL *dev_loc, REAL *dev_diag_mass){
4.    int i,j,k;
5.    REAL tmp, coef;
6.
7.    for (i=blockIdx.x*blockDim.x+threadIdx.x; i<nrparticles; i+=blockDim.x * gridDim.x) {
8.      coef=dev_penalty*(dev_Cxadj[i+1]-dev_Cxadj[i]);
9.  #pragma unroll
10.     for (k=0; k<6; k++){
11.       tmp=dev_forces[6*i+k];
12.       for (j=dev_Cxadj[i]; j<dev_Cxadj[i+1]; j++)
13.         tmp+=dev_loc[dev_Cadjncy[j]+k];
14.
15.       dev_displacements[6*i+k]=tmp/(coef+dev_diag_mass[6*i+k]);
16.     }
17.   }
18. }
```

Implementação 2: sem mecanismos de sincronização e leitura de dados para duas variáveis locais.

```
1.  __global__ void GlobalSol(int nrparticles, int *dev_Cxadj, int *dev_Cadjncy,
2.          REAL *dev_displacements, REAL *dev_forces,
3.          REAL *dev_loc,  REAL *dev_diag_mass){
4.    int i,j,k;
5.    REAL tmp, coef;
6.    // uso de vars locais
7.    int cxadj_i, cxadj_i1;
8.
9.    for (i=blockIdx.x*blockDim.x+threadIdx.x; i<nrparticles; i+=blockDim.x * gridDim.x) {
10.     cxadj_i= dev_Cxadj[i];
11.     cxadj_i1= dev_Cxadj[i+1];
12.     coef=dev_penalty*(cxadj_i1 - cxadj_i);
13.
14. #pragma unroll
15.     for (k=0; k<6; k++){
16.       tmp=dev_forces[6*i+k];
17.       for (j=cxadj_i; j<cxadj_i1; j++)
18.         tmp+=dev_loc[dev_Cadjncy[j]+k];
19.       dev_displacements[6*i+k]=tmp/(coef+dev_diag_mass[6*i+k]);
20.     }
21.   }
22. }
```

Implementação 3: sem mecanismos de sincronização e leitura de dados para duas variáveis na memória partilhada.

```
1.  __global__ void GlobalSol(int nrparticles, int *dev_Cxadj, int *dev_Cadjncy,
2.              REAL *dev_displacements, REAL *dev_forces,
3.              REAL *dev_loc, REAL *dev_diag_mass) {
4.    int i,j,k;
5.    REAL tmp, coef;
6.    // uso de shared memory
7.    extern __shared__ int xadj[];
8.
9.    for (i=blockIdx.x*blockDim.x+threadIdx.x; i<nrparticles; i+=blockDim.x * gridDim.x) {
10.     xadj[threadIdx.x]=dev_Cxadj[i];
11.     xadj[threadIdx.x+1]=dev_Cxadj[i+1];
12.     coef=dev_penalty*(xadj[threadIdx.x+1]-xadj[threadIdx.x]);
13. #pragma unroll
14.     for (k=0; k<6; k++){
15.       tmp=dev_forces[6*i+k];
16.       for (j=xadj[threadIdx.x]; j<xadj[threadIdx.x+1]; j++)
17.         tmp+=dev_loc[dev_Cadjncy[j]+k];
18.       dev_displacements[6*i+k]=tmp/(coef+dev_diag_mass[6*i+k]);
19.     }
20.   }
21. }
```

Implementação 4: novo *kernel*, sem mecanismos de sincronização e acesso direto à memória global.

```
1.  __global__ void GlobalSol(int nrparticles, int *dev_Cxadj, int *dev_Cadjncy,
2.                            REAL *dev_displacements, REAL *dev_forces,
3.                            REAL *dev_loc, REAL *dev_diag_mass) {
4.    int i,j,k;
5.    REAL tmp, coef;
6.    for (i=blockIdx.x*blockDim.x+threadIdx.x; i<nrparticles; i+=blockDim.x * gridDim.x) {
7.      coef=dev_penalty*(dev_Cxadj[i+1]-dev_Cxadj[i]);
8.  #pragma unroll
9.      for (k=0; k<6; k++){
10.       tmp=dev_forces[6*i+k];
11.       for (j=dev_Cxadj[i]; j<dev_Cxadj[i+1]; j++)
12.         tmp+=dev_loc[dev_Cadjncy[j]+k];
13.       dev_displacements[6*i+k]=tmp/(coef+dev_diag_mass[6*i+k]);
14.     }
15.   }
16. }
```

Implementação 5: novo *kernel*, sem mecanismos de sincronização e leitura de dados para duas variáveis locais.

```
1.  __global__ void GlobalSol(int nrparticles, int *dev_Cxadj, int *dev_Cadjncy,
2.          REAL *dev_displacements, REAL *dev_forces,
3.          REAL *dev_loc,  REAL *dev_diag_mass){
4.    int i,j,k;
5.    REAL tmp, coef;
6.    // uso de vars locais
7.    int cxadj_i, cxadj_i1;
8.
9.    for (i=blockIdx.x*blockDim.x+threadIdx.x; i<nrparticles; i+=blockDim.x * gridDim.x) {
10.     cxadj_i= dev_Cxadj[i];
11.     cxadj_i1= dev_Cxadj[i+1];
12.     coef=dev_penalty*(cxadj_i1 - cxadj_i);
13. #pragma unroll
14.     for (k=0; k<6; k++){
15.       tmp=dev_forces[6*i+k];
16.       for (j=cxadj_i; j<cxadj_i1; j++)
17.         tmp+=dev_loc[dev_Cadjncy[j]+k];
18.       dev_displacements[6*i+k]=tmp/(coef+dev_diag_mass[6*i+k]);
19.     }
20.   }
21. }
```

Implementação 6: novo *kernel*, sem mecanismos de sincronização e leitura de dados para duas variáveis na memória partilhada.

```
1.  __global__ void GlobalSol(int nrparticles, int *dev_Cxadj, int *dev_Cadjncy,
2.                            REAL *dev_displacements, REAL *dev_forces,
3.                            REAL *dev_loc, REAL *dev_diag_mass){
4.    int i,j,k;
5.    REAL tmp, coef;
6.    // uso de shared memory
7.    extern __shared__ int xadj[];
8.
9.    for (i=blockIdx.x*blockDim.x+threadIdx.x; i<nrparticles; i+=blockDim.x * gridDim.x) {
10.     xadj[threadIdx.x]=dev_Cxadj[i];
11.     xadj[threadIdx.x+1]=dev_Cxadj[i+1];
12.     coef=dev_penalty*(xadj[threadIdx.x+1]-xadj[threadIdx.x]);
13. #pragma unroll
14.     for (k=0; k<6; k++){
15.       tmp=dev_forces[6*i+k];
16.       for (j=xadj[threadIdx.x]; j<xadj[threadIdx.x+1]; j++)
17.         tmp+=dev_loc[dev_Cadjncy[j]+k];
18.       dev_displacements[6*i+k]=tmp/(coef+dev_diag_mass[6*i+k]);
19.     }
20.   }
21. }
```

Kernel CalcA

Implementação 1: implementação utilizando as bibliotecas cuBLAS e cuSparse

```
1.  …
2.  #ifdef CUBLASSPARSE
3.    initBLAS();
4.    initcuSparse();
5.    //array para usar no CalcA
6.    cudaMalloc((REAL **)&d_tmp, sizeof(REAL)*(6*(2*nrcontacts-nrwallcontacts)));
7.    callCusparseFun(contactslist);
8.  #endif
9.  …
```

```
1.  void initBLAS(){
2.      //* Get handle to the CUBLAS context */
3.      //cublasHandle_t handleBlas=0; //definido em data.h
4.      handleBlas=0;
5.      cublasCreate(&handleBlas);
6.      //BLAS operations anywhere
7.      //cublasDestroy(handleBlas);  //colocado no cleandevmem;
8.  }
```

```
1.  void initcuSparse(){
2.      /* Get handle to the CUSPARSE context */
3.      //cusparseHandle_t cusparseHandle = 0; definido em data.h
4.      cusparseHandle = 0;
5.      //cusparseStatus_t cusparseStatus;   //definido em data.h
6.      cusparseStatus = cusparseCreate(&cusparseHandle);
7.      //checkCudaErrors(cusparseStatus);
8.
9.      //cusparseMatDescr_t descr = 0;  definido em data.h
10.     descr = 0;
11.     cusparseStatus = cusparseCreateMatDescr(&descr);
12.     //checkCudaErrors(cusparseStatus);
13.
14.     cusparseSetMatType(descr,CUSPARSE_MATRIX_TYPE_GENERAL);
15.     cusparseSetMatIndexBase(descr,CUSPARSE_INDEX_BASE_ZERO);
16. }
```

```
1.  void create_particles2contacts_matrix (int *contactslist, int *h_cooRowIndA,
2.                                          int *h_cooColIndA, REAL *h_cooValA, int *count_a){
3.    int  line, c1,c2;
4.    int i,j;
5.    int count=0;
6.    for (i=0; i<nrcontacts-nrwallcontacts; i++){
7.        c1=6*(contactslist[2*i]-1);
8.        c2=6*(contactslist[2*i+1]-1);
9.        line=12*i;
10.       for(j=0; j<6; j++){
11.       h_cooValA[line]=1.;
12.       h_cooColIndA[line]=c1++;
13.       h_cooRowIndA[line]=line;
14.      h_cooValA[line+6]=1.;
15.      h_cooColIndA[line+6]=c2++;
16.      h_cooRowIndA[line+6]=6+line++;
17.     }
18.   }
19.   for (i=0; i<nrwallcontacts; i++){
20.       c1=6*(contactslist[2*i]-1);
21.       line=12*(nrcontacts-nrwallcontacts)+6*i;
22.       for(j=0; j<6; j++){
23.            h_cooValA[line]=1.;
24.            h_cooColIndA[line]=c1++;
25.            h_cooRowIndA[line]=line++;
26.        }
27.    }
28.   *count_a = line;
29. }
```

```
1.  void callCusparseFun (int *contactslist){
2.
3.      //int nnz = 0; //nnz - The number of nonzero elements in the matrix. definido em data.h
4.      nnz = 0;
5.      //host
6.      int *h_cooRowIndA, *h_cooColIndA;
7.      REAL *h_cooValA;
8.      //Device
9.      //int *d_cooColIndA, *d_cooRowIndA, *d_csrRowPtrA;  //device
10.     //REAL *d_cooValA;
11.
12.     int n = 6 * nrparticles; //n - The number of columns in the matrix.
13.     //int m = 12*(nrcontacts-nrwallcontacts)-6*nrwallcontacts;//// m - The number of rows in
    the matrix.
14.     int m = 12*(nrcontacts-nrwallcontacts)+6*nrwallcontacts;//// m - The number of rows in the
    matrix.
15.     //Alloc host memory
16.     h_cooRowIndA= (int *)malloc(sizeof(int)*6*(2*nrcontacts-nrwallcontacts));
17.     h_cooColIndA= (int *)malloc(sizeof(int)*6*(2*nrcontacts-nrwallcontacts));
18.     h_cooValA= (REAL *)malloc(sizeof(REAL)*6*(2*nrcontacts-nrwallcontacts));
19.    //Criar a matrix formato coo
20.     create_particles2contacts_matrix (contactslist, h_cooRowIndA, h_cooColIndA, h_cooValA,
    &nnz);
21.     //device_matrix
22.     cudaMalloc((void **)&d_cooRowIndA, sizeof(int)*nnz);
23.     cudaMalloc((void **)&d_cooColIndA, sizeof(int)*nnz);
24.     cudaMalloc((void **)&d_cooValA, nnz * sizeof(REAL));
25.     cudaMemcpy(d_cooRowIndA, h_cooRowIndA, sizeof(int)*nnz, cudaMemcpyHostToDevice);
26.     cudaMemcpy(d_cooColIndA, h_cooColIndA, sizeof(int)*nnz, cudaMemcpyHostToDevice);
27.     cudaMemcpy(d_cooValA, h_cooValA, sizeof(REAL)*nnz, cudaMemcpyHostToDevice);
28.     /* conversion routines (convert matrix from COO 2 CSR format) */
29.     cudaMalloc((void**)&d_csrRowPtrA,sizeof(int)*(m+1));
30.     cusparseStatus =cusparseXcoo2csr(cusparseHandle, d_cooRowIndA, nnz, m+1, d_csrRowPtrA,
    CUSPARSE_INDEX_BASE_ZERO);
31.   free(h_cooRowIndA);
32.   free(h_cooColIndA);
33.   free(h_cooValA);
34.   cudaFree(d_cooRowIndA);
35. }
```

Anexo C: Trabalho realizado sobre os kernels especializados

Dado que foi seguida a mesma metodologia anterior serão apresentadas tabelas únicas para cada *kernel* analisado.

Kernel UpdateWall

Tabela C.1: Ensaios realizados para o *kernel* Update_wall

#Block	#Threads/B	Reg	ShMem (B)	Occupancy (%)	Time	Calls	Avg (us)	Speedup	Obs.
1*NSMs,	1024	23	4096	67	1,0452ms	10	104,52	0,4%	
3*NSMs,	512	23	2048	67	1,0492ms	10	104,92		ORIGINAL
6*NSMs,	256	23	1024	83	972,15us	10	97,215	7,3%	
12*NSMs,	128	23	512	67	963,48us	10	96,348	8,2%	
24*NSMs,	64	23	256	33	911,18us	10	91,117	13,2%	
48*NSMs,	32	23	128	17	685,20us	10	68,52	34,7%	
48*NSMs,	32	20	128	17	702.59us	10	70,259	33,0%	limitado # registos = 20
48*NSMs,	32	23	128	17	515.62us	10	51,561	50,9%	cudaFuncCachePreferL1
48*NSMs,	32	23	128	17	514.63us	10	51,463	51,0%	cudaFuncCachePreferL1

Kernel GlobalSol

Tabela C.2: Ensaios realizados para o *kernel* GlobalSol0

#Block	#Threads/B	Reg	ShMem (B)	Occupancy (%)	Time (ms)	Calls	Avg (ms)	Speedup	Observações
1*NSMs,	1024	13	4100	67	1,0868	1	1,0868	31,5%	
2*NSMs,	768	13	3076	100	1,064	1	1,064	33,0%	
3*NSMs,	512	13	2052	100	1,063	1	1,063	33,0%	
2*NSMs,	512	13	2052	100	1,5872	1	1,5872		ORIGINAL
4*NSMs,	384	13	1540	100	1,0627	1	1,0627	33,0%	
6*NSMs,	256	13	1028	100	1,0622	1	1,0622	33,1%	
6*NSMs	256	13	1028	100	1,0659	1	1,0659	32,8%	cudaFuncCachePreferL1
6*NSMs	256	13	1028	100	1,0655	1	1,0655	32,9%	cudaFuncCachePreferL1
8*NSMs	192	13	772	100	1,0632	1	1,0632	33,0%	
12*NSMs,	128	13	516	67	1,2249	1	1,2249	22,8%	
24*NSMs,	64	13	260	33	1,4595	1	1,4595	8,0%	
48*NSMs,	32	13	132	17	2,6599	1	2,6599	-67,6%	

Kernel CalcAwall

Tabela C.3: Ensaios realizados para o *kernel* CalcAwall

#Block	#Threads/B	Reg	ShMem (B)	Occupancy (%)	Time (ms)	Calls	Avg (us)	Speedup	Obs.
CalcAwall	800	28	48000	52	1,7627	20	88,134		ORIGINAL
CalcAwall	800	28	48000	52	1,7673	20	88,363	-0,3%	
3*NSMs,	512	28	30720	33	1,8411	20	92,055	-4,4%	
4*NSMs	384	28	23040	50	1,7449	20	87,243	1,0%	
6*NSMs,	256	28	15360	50	1,7331	20	86,653	1,7%	
9*NSMs	160	28	9600	52	1,7676	20	88,379	-0,3%	
12*NSMs,	128	28	7680	50	1,7457	20	87,285	1,0%	
12*NSMs,	128	28	7680	50	1,341	20	67,049	23,9%	cudaFuncCachePreferL1
24*NSMs,	64	28	3840	33	1,6414	20	82,068	6,9%	
24*NSMs,	64	28	3840	33	1,3361	20	66,804	24,2%	cudaFuncCachePreferL1
48*NSMs,	32	28	1920	17	1,5292	20	76,459	13,2%	
48*NSMs,	32	28	1920	17	1,5362	20	76,81	12,8%	limitado a 20 registos
48*NSMs,	32	28	1920	17	1,3191	20	65,953	25,2%	cudaFuncCachePreferL1

Kernel CalcA2

Tabela C.4: Ensaios realizados para o *kernel* CalcA2

#Block	#Threads/B	Reg	ShMem (B)	Occupancy (%)	Time (s)	Calls	Avg (ms)	Speedup	Obs.
3*NSMs,	512	58	43008	33	1,66169	20	83,085	5,7%	
1*NSMs,	512	58	43008	33	1,76235	20	88,118		ORIGINAL
6*NSMs,	256	58	21504	33	1,68858	20	84,429	4,2%	
8*NSMs	192	58	16128	25	1,41877	20	70,939	19,5%	
8*NSMs	192	58	16128	25	2,7616	20	138,08	-56,7%	limitado a 20 registos
8*NSMs	192	58	16128	15	1,81999	20	90,999	-3,3%	cudaFuncCachePreferL1
12*NSMs,	128	58	10752	33	1,71888	20	85,944	2,5%	
24*NSMs,	64	58	5376	33	1,73644	20	86,822	1,5%	
48*NSMs,	32	58	2688	17	1,65952	20	82,976	5,8%	

Kernel CalcA2wall

Tabela C.5: Ensaios realizados para o *kernel* CalcA2wall

#Block	#Threads/B	Reg	ShMem (B)	Occupancy (%)	Time (ms)	Calls	Avg (us)	Speedup	Obs.
1*NSMs,	704	43	42240	46	2,4183	20	120,91		ORIGINAL
3*NSMs,	512	43	30720	33	1,883	20	94,149	22,1%	
1*NSMs,	512	43	30720	33	1,7395	20	86,972	28,1%	
4*NSMs	352	43	21120	46	2,157	20	107,85	10,8%	
6*NSMs,	256	43	15360	33	1,7931	20	89,656	25,8%	
6*NSMs,	224	43	13440	44	2,1172	20	105,86	12,4%	
8*NSMs	192	43	11520	38	2,0142	20	100,71	16,7%	
12*NSMs,	128	43	7680	42	1,9368	20	96,84	19,9%	
24*NSMs,	64	43	3840	33	1,696	20	84,797	29,9%	
48*NSMs,	32	43	1920	17	1,5938	20	79,691	34,1%	
48*NSMs,	32	43	1920	17	1,7501	20	87,503	27,6%	limitado a 20 registos
48*NSMs,	32	43	1920	17	1,3717	20	68,586	43,3%	cudaFuncCachePreferL1

kernel Local2Global_wall

Tabela C.6: Ensaios realizados ao *kernel* Local2Global_wall

#Block	#Threads/B	Reg	ShMem (B)	Occupancy (%)	Time (us)	Calls	Avg (us)	Speedup	Obs.
1*NSMs,	800	30	48000	52	236,89	10	23,689		ORIGINAL
3*NSMs,	512	30	30720	33	284,72	10	28,472	-20,2%	
4*NSMs	384	30	23040	50	232,75	10	23,275	1,7%	
6*NSMs,	256	30	15360	50	232,96	10	23,295	1,7%	
6*NSMs,	256	30	15360	50	281,69	10	28,169	-18,9%	limitado a 20 registos
6*NSMs,	256	30	15360	50	283,77	10	28,377	-19,8%	cudaFuncCachePreferL1
12*NSMs,	128	30	7680	50	234,26	10	23,425	1,1%	
24*NSMs,	64	30	3840	33	248,05	10	24,805	-4,7%	
48*NSMs,	32	30	1920	17	245,88	10	24,587	-3,8%	

Kernel Local2Global2

Tabela C.7: Ensaios realizados para o *kernel* Local2Global2

#Block	#Threads/B	Reg	ShMem (B)	Occupancy (%)	Time (ms)	Calls	Avg (ms)	Speedup	Obs.
3*NSMs,	512	63	43008	33	704,22	10	70,422	1,6%	
1*NSMs,	512	63	43008	33	715,32	10	71,532		ORIGINAL
6*NSMs,	256	63	21504	33	704,57	10	70,457	1,5%	
8*NSMs	192	63	16128	33	626,04	10	62,604	12,5%	
8*NSMs	192	63	16128	33	2759,42	20	137,97	-92,9%	limitado a 20 registos
8*NSMs	192	63	16128	33	711,55	10	71,155	0,5%	cudaFuncCachePreferL1
12*NSMs,	128	63	10752	33	706,78	10	70,678	1,2%	
24*NSMs,	64	63	5376	33	709,87	10	70,987	0,8%	
48*NSMs,	32	63	2688	17	654	10	65,4	8,6%	
48*NSMs,	32	63	2688	17	708,070	10	70,807	1,0%	cudaFuncCachePreferL1

Kernel Local2Global2wall

Tabela C.8: Ensaios realizados para o *kernel* Local2Global2_wall

#Block	#Threads/B	Reg	ShMem (B)	Occupancy (%)	Time (us)	Calls	Avg (us)	Speedup	Obs.
1*NSMs,	800	31	48000	52	235,29	10	23,529		ORIGINAL
3*NSMs,	512	31	30720	33	297,79	10	29,779	-26,6%	
1*NSMs,	512	31	30720	33	274,25	10	27,424	-16,6%	
4*NSMs	384	31	23040	50	225,35	10	22,534	4,2%	
4*NSMs	384	31	23040	50	281,45	10	28,144	-19,6%	limitado a 20 registos
4*NSMs	384	31	23040	50	226,09	10	22,608	3,9%	cudaFuncCachePreferL1
6*NSMs,	256	31	15360	50	226,25	10	22,625	3,8%	
6*NSMs,	256	31	15360	50	340,01	10	34	-44,5%	cudaFuncCachePreferL1
9*NSMs	160	31	9600	52	231,25	10	23,125	1,7%	
12*NSMs,	128	31	7680	50	226,82	10	22,682	3,6%	
12*NSMs,	128	31	7680	50	309,18	10	30,917	-31,4%	cudaFuncCachePreferL1
24*NSMs,	64	31	3840	33	291,93	10	29,192	-24,1%	
48*NSMs,	32	31	1920	17	292,48	10	29,247	-24,3%	

Kernel LocalMin_wall

Tabela C.9: Ensaios realizados para o *kernel* LocalMin_wall

#Block	#Threads/B	Reg	ShMem (B)	Occupancy (%)	Time (us)	Calls	Avg (us)	Speedup	Obs.
1*NSMs,	768	38	46080	50	298,32	10	29,831		ORIGINAL
1*NSMs,	768	20	46080	50	362,18	10	36,218	-21,4%	limite de 20 registos
3*NSMs,	512	38	30720	33	311,44	10	31,143	-4,4%	
4+NSMs	384	38	23040	50	253,02	10	25,302	15,2%	
6*NSMs,	256	38	15360	50	251,86	10	25,186	15,6%	
8*NSMs	192	38	11520	50	252,67	10	25,266	15,3%	
12*NSMs,	128	38	7680	50	251,75	10	25,175	15,6%	
12*NSMs,	128	20	7680	50	344,22	10	34,421	-15,4%	limite de 20 registos
12*NSMs,	128	38	7680	50	362,9	10	36,29	-21,7%	cudaFuncCachePreferL1
16*NSMs	96	38	5760	50	253,25	10	25,325	15,1%	
24*NSMs,	64	38	3840	33	263,49	10	26,348	-11,7%	
48*NSMs,	32	38	1920	17	353,04	10	35,304	-18,3%	

Kernel LocalMin0

Tabela C.10: Ensaios realizados para o *kernel* LocalMin0

#Block	#Threads/B	Reg	ShMem (B)	Occupancy (%)	Time (ms)	Calls	Avg (ms)	Speedup	Obs.
1*NSMs,	1024	9	0	67	36,105	10	3,6105	0,1%	
1*NSMs,	1024	9	0	67	36,141	10	3,6141	0,0%	cudaFuncCachePreferL1
1*NSMs,	768	9	0	100	36,134	10	3,6134	0,0%	
3*NSMs,	512	9	0	100	36,134	10	3,6134		ORIGINAL
3*NSMs,	512	9	0	100	36,097	10	3,6097	0,1%	cudaFuncCachePreferL1
4*NSMs,	384	9	0	100	36,166	10	3,6166	-0,1%	
6*NSMs,	256	9	0	100	36,13	10	3,613	0,0%	
8*NSMs	192	9	0	100	36,165	10	3,6165	-0,1%	
12*NSMs,	128	9	0	67	36,134	10	3,6134	0,0%	
24*NSMs,	64	9	0	33	36,187	10	3,6187	-0,1%	
48*NSMs,	32	9	0	17	36,205	10	3,6205	-0,2%	

Kernel LocalMin0_wall

Tabela C.11: Ensaios realizados para ao *kernel* LocalMin0_wall

#Block	#Threads/B	Reg	ShMem (B)	Occupancy (%)	Time (us)	Calls	Avg (us)	Speedup	Obs.
1*NSMs,	1024	9	0	67	50,327	10	5,032	11,7%	
1*NSMs,	768	9	0	100	61,584	10	6,158	-8,1%	
3*NSMs,	512	9	0	100	56,962	10	5,696		ORIGINAL
4*NSMs,	384	9	0	100	54,06	10	5,406	5,1%	
6*NSMs,	256	9	0	100	58,012	10	5,801	-1,8%	
8*NSMs	192	9	0	100	55,25	10	5,525	3,0%	
12*NSMs,	128	9	0	67	48,44	10	4,844	15,0%	
12*NSMs,	128	9	0	67	48,059	10	4,805	15,6%	cudaFuncCachePreferL1
24*NSMs,	64	9	0	33	57,584	10	5,758	-1,1%	
48*NSMs,	32	9	0	17	84,406	10	8,44	-48,2%	

Kernel LocalMin1

Tabela C.12: Ensaios realizados para o *kernel* LocalMin1

#Block	#Threads/B	Reg	ShMem (B)	Occupancy (%)	Time (ms)	Calls	Avg (ms)	Speedup	Obs.
1*NSMs,	1024	27	16384	67	134,35	10	13,435	7,2%	
2*NSMs	576	27	9216	75	135,78	10	13,578	6,2%	
3*NSMs,	512	27	8192	67	146,97	10	14,697	-1,5%	
3*NSMs,	448	27	7168	58	144,79	10	14,479		ORIGINAL
3*NSMs,	448	20	7168	58	152,78	10	15,278	-5,5%	limite 20 registos
4*NSMs,	384	27	6144	75	135,6	10	13,56	6,3%	
5*NSMs	288	27	4608	75	152,18	10	15,218	-5,1%	
6*NSMs,	256	27	4096	67	148,2	10	14,82	-2,4%	
12*NSMs,	128	27	2048	67	147,14	10	14,714	1,6%	
24*NSMs,	64	27	1024	33	117,18	10	11,718	19,1%	
24*NSMs,	64	20	1024	33	130,47	10	13,047	9,9%	limite 20 registos
24*NSMs,	64	27	1024	33	111,06	10	11,106	23,3%	cudaFuncCachePreferL1
48*NSMs,	32	27	512	17	139,79	10	13,979	3,5%	

Kernel Local2Min1_wall

Tabela C.13: Ensaios realizados para o *kernel* LocalMin1_wall

#Block	#Threads/B	Reg	ShMem (B)	Occupancy (%)	Time (us)	Calls	Avg (us)	Speedup	Obs.
1*NSMs,	1024	26	16384	67	169,7	10	16,969	-5,6%	
2*NSMs	608	26	9728	79	196,48	10	19,648	-22,3%	
3*NSMs,	512	26	8192	67	175,79	10	17,578	-9,4%	
3*NSMs,	448	26	7168	58	160,7	10	16,07		ORIGINAL
3*NSMs,	448	20	7168	58	195,56	10	19,555	-21,7%	limite 20 registos
4*NSMs,	384	26	6144	75	154,82	10	15,481	3,7%	
5*NSMs	288	26	4608	75	163,62	10	16,361	-1,8%	
6*NSMs,	256	26	4096	67	153,53	10	15,352	4,5%	
6*NSMs,	256	20	4096	67	214,39	10	21,439	-33,4%	limite 20 registos
6*NSMs,	256	26	4096	67	153,2	10	15,32	4,7%	cudaFuncCachePreferL1
12*NSMs,	128	26	2048	67	156,6	10	15,66	2,6%	
24*NSMs,	64	26	1024	33	171,31	10	17,131	-6,6%	
48*NSMs,	32	26	512	17	202,98	10	20,298	-26,3%	

Kernel LocalRHS

Tabela C.14: Ensaios realizados para o *kernel* LocalRHS

#Block	#Threads/B	Reg	ShMem (B)	Occupancy (%)	Time (ms)	Calls	Avg (ms)	Speedup	Obs.
1*NSMs,	1024	11	0	67	88,997	9	9,8885	-0,1%	
1*NSMs,	1024	11	0	67	88,862	9	9,8735	0,0%	cudaFuncCachePreferL1
2*NSMs,	768	11	0	100	88,984	9	9,8871	-0,1%	
3*NSMs,	512	11	0	100	88,982	9	9,8868	-0,1%	
2*NSMs,	512	11	0	100	88,886	9	9,8762		ORIGINAL
2*NSMs,	512	11	0	100	88,97	9	9,8855	-0,1%	cudaFuncCachePreferL1
4*NSMs,	384	11	0	100	89,014	9	9,8905	-0,1%	
6*NSMs,	256	11	0	100	89,031	9	9,8924	-0,2%	
8*NSMs	192	11	0	100	88,921	9	9,8801	0,0%	
12*NSMs,	128	11	0	67	88,861	9	9,8734	0,0%	
24*NSMs,	64	11	0	33	88,961	9	9,8846	-0,1%	
48*NSMs,	32	11	0	17	88,899	9	9,8777	0,0%	

Kernel get Overlap

Tabela C..15: Ensaios realizados para o *kernel* getOverlap

#Block	#Threads/B	Reg	ShMem (B)	Occupancy (%)	Time (ms)	Calls	Avg	Speedup	Obs.
1*NSMs,	1024	25	4096	67	118,56	10	11,856	-0,5%	
3*NSMs,	512	25	2048	67	130,7	10	13,070	-10,8%	
2*NSMs,	512	25	2048	67	118	10	11,800		ORIGINAL
2*NSMs,	512	20	2048	67	150,39	10	15,039	-27,4%	limite 20 registos
1*NSMs,	512	25	2048	67	174,46	10	17,446	-47,8%	
6*NSMs,	256	25	1024	67	130,5	10	13,050	-10,6%	
12*NSMs,	128	25	512	67	126,81	10	12,681	-7,5%	
24*NSMs,	64	25	256	33	103,62	10	10,362	12,2%	
24*NSMs,	64	25	256	33	103,1	10	10,310	12,6%	cudaFuncCachePreferL1
24*NSMs,	64	25	256	33	103,92	10	10,392	11,9%	
48*NSMs,	32	25	128	17	105,82	10	10,582	10,3%	

Kernel normals3D

Tabela C.16: Ensaios realizados para o *kernels* normals3D

#Block	#Threads/B	Reg	ShMem (B)	Occupancy (%)	Time (ms)	Calls	Avg (ms)	Speedup	Obs.
1*NSMs,	1024	24	12288	67	25,069	1	25,069	-6,3%	
2*NSMs	672	24	8064	88	27,015	1	27,015	-14,5%	
3*NSMs,	512	24	6144	67	23,587	1	23,587		ORIGINAL
3*NSMs,	512	20	6144	67	27,137	1	27,137	-15,1%	limite 20 registos
3*NSMs,	448	24	5376	88	26,242	1	26,242	-11,3%	
6*NSMs,	256	24	3072	83	25,407	1	25,407	-7,7%	
8*NSMs	192	24	2304	88	25,758	1	25,758	-9,2%	
12*NSMs,	128	24	1536	67	24,539	1	24,539	-4,0%	
24*NSMs,	64	24	768	33	20,75	1	20,75	12,0%	
48*NSMs,	32	24	384	17	19,924	1	19,924	15,5%	
48*NSMs,	32	20	384	17	700,74	10	70,073	-197,1%	limite 20 registos
48*NSMs,	32	24	384	17	20,027	1	20,027	15,1%	cudaFuncCachePreferL1

Kernel normalsWalls3D

Tabela C.17: Ensaios realizados para o *kernel* normalwalls3D

#Block	#Threads/B	Reg	ShMem (B)	Occupancy (%)	Time (us)	Calls	Avg (us)	Speedup	Obs.
1*NSMs,	896	36	32256	58	39,227	1	39,227	-14,2%	
3*NSMs,	512	36	18432	33	44,74	1	44,74	-30,3%	
3*NSMs,	448	36	16128	58	36,788	1	36,788	-7,1%	
2*NSMs	384	36	13824	50	34,349	1	34,349		ORIGINAL
2*NSMs	384	20	13824	50	59,637	1	59,637	-73,6%	limite 20 registos
6*NSMs,	256	36	9216	50	32,774	1	32,774	4,6%	
6*NSMs,	256	20	9216	50	49,142	1	49,142	-43,1%	limite 20 registos
6*NSMs,	256	36	9216	50	32,452	1	32,452	5,5%	
6*NSMs,	256	36	9216	50	33,135	1	33,135	3,5%	cudaFuncCachePreferL2
8*NSMs	192	36	6912	50	33,195	1	33,195	3,4%	
12*NSMs,	128	36	4608	58	33,076	1	33,076	3,7%	
24*NSMs,	64	36	2304	33	36,451	1	36,451	-6,1%	
48*NSMs,	32	36	1152	17	42,258	1	42,258	-23,0%	